

|| Programmer's Mind. ||

Advanced Features (Core Series) - Updated To **Java 8**.

Introduction to Java Programming

◇ *Easy Version* ◇

- Harry. H. Chaudhary.

∞ Essential Java Skills - Made Easy ! ∞

SECOND EDITION



Introduction to Java Programming

Advanced Features (Core Series)
Updated To Java 8.



-Harry.H.Chaudhary.

(IT Manager & Anonymous Hactivist @ Anonymous International)



Publisher's Note:

Every possible effort has been made to ensure that the information contained in this book is accurate, and the publisher or the Author–Harry. H. Chaudhary can't accept responsibility for any errors or omissions, however caused. All liability for loss, disappointment, negligence or other damage caused by the reliance of the Technical Programming or other information contained in this book, of in the event of bankruptcy or liquidation or cessation of trade of any company, individual; or firm mentioned, is hereby excluded.

Sun Microsystems and Oracle the trademarks, are the Trademarks of the Sun (Now Oracle) & Oracle group of companies. Sun Microsystems and Oracle the trademarks are listed at their websites. All other marks are property of their respective owners. The examples of companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

The author and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

This book expresses the author views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, and Publisher, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Copyright © 2010-2014. By Harry. H. Chaudhary (CEO Programmers Mind.)

Published By Programmers Mind || Createspace Inc. OD Publishing, LLC USA.

All rights reserved. This book or any portion thereof may not be reproduced or used in any manner whatsoever without the express written permission of the author except for the use of brief quotations in a book review or scholarly journal.

ISBN-13: 978-1500864514.

ISBN-10: 1500862347.

Printed By Createspace O-D Publishing LLC USA. [SECOND EDITION 2014]

**Marketing & Distributed By || Amazon Inc.|| Programmer's Mind Inc. || Lulu.com
|| Google Books & Google Play Store. || other 25 worldwide Bookstores.**

Dedication

“This book is dedicated to all those who make the daily sacrifices,
Especially those who’ve made sacrifice, to ensure our freedom & security.”

Thanks to Lord Shiva a lot for giving me the technical and all abilities in
my life to write.

Dear Dad, Thank you baauji, for teaching me by example what it means
to live a life based on principles.

Dear 2 Mom’s, Thank you for showing me in a real way how to live
according to the most imp. principle, and unconditional love.

Dear Sisters & Brother+Priya, Thank U, your smile brightens my every
day. Your zest makes my heart sing. I love you All.

I would especially like to mention the name of beautiful faces inside my
life who helped me in coping with my sorrows:

Thank you Priyanka, you are the meaning of my life and apple of my
eyes, I Love You more than I can say.

Thank U Hem Zizu, Navneet, Aman(Rajjo) Eminem - you are the
inspiration you made me like “Sing for the movement” ,

Thanks to all Anonymous And Black Hat Hackers worldwide.

In Loving Memories of My Loved One –My Uncle Lt. G.C

In Loving Memories of My Loved One –My Lt. Grand Mom.

You told me that everything will be okay in the end,

You also told me that, if it's not okay, it's not the end.

“I'll search for you through 1000 worlds & 10000 lifetimes until I find you

About Author:

Harry, is an **Anonymous Hactivist**, GOC Famous computer Programmer and Bestselling Java Author and **scientifically Hacking Professional** has a unique experience in the field of computers Programming, **Hacking and Cyber Security**.

He has helped many Countries Governments and many multinational Software companies of around the globe to secure their networks and securities. He has authored several books on Various Computers Programming Languages and computer security & Hacking.

He is technically graduate software engineer and Master. He is the leading authority on **C** Programming and **C++** Programming as well as on **Core Java** and **Data Structure and Algorithms**. His acclaimed **C** and **C++**, **C#** & **Java** books. He has over 5 years of experience as a software methodologist. His teaching and research interests are in the areas of artificial intelligence, programming languages.

He is living two lives. One life, He is a Computer program writer for a respectable software company. The other life is lived in computers, where he go by the hacker alias **“Chief Hacker – Captain Harry”**. Currently he is working as offline IT manager @ world famous community **Anonymous international Community**.

-Team Anonymous.

Author side :

You may have noticed something missing here: no impressive of credentials. I haven't been a professor at a Prestigious University for a quarter-century; neither am I a top executive at a Silicon Valley giant. In some ways, I'm a student of Technology, just like you are.

And my experience over the years has shown me that many of the people who know the most about how technology works also have rather limited success in explaining what they know in a way that will allow me to understand it. My interests, and I believe my skills, lie not in being an expert, but an educator, in presenting complex information in a form that is sensible, digestible and fun to read my books.

“What is real? How do you define *real*? If you're talking about what you can feel, what you can smell, what you can taste and see, then real is simply, electrical signals interpreted by your brain.”

“... I am just now beginning to discover the difficulty of expressing one's ideas on paper. As long as it consists solely of description it is pretty easy; but where reasoning

comes into play, to make a proper connection, a clearness & a moderate fluency, is to me, as I have said, a difficulty of which I had no idea ...”

– *Harry*.

∞ Inside Chapters at a Glance ∞

Unit	Chapters & Topics Inside the Book	Page
00.	Preface.	006.
01.	Overview of Java	008.
02.	Java Language	023.
03.	Control Statements	039.
04.	Scanner class, Arrays & Command Line Args	048.
05.	Class & Objects in Java	059.
06.	Inheritance in Java	082.
07.	Object oriented programming	098.
08.	Packages in Java	106.
09.	Interface in Java	115.
10.	String and StringBuffer	129.
11.	Exception Handling	142.
12.	Multi-Threaded Programming	185.
13.	Modifiers/Visibility modes	240.
14.	Wrapper Class	255.
15.	Input/Output in Java	273.
16.	Applet Fundamentals	338.
17.	Abstract Windows Toolkit (AWT)	357.
18.	Introducton To AWT Events	404.
19.	Painting in AWT	445.

20.	java.lang.Object Class	470.
21.	Collection Framework	490.
22.	Java 8 Features for Developers – Lambdas.	540.
23.	Java 8 Functional interface,Stream & Time API.	565.
24.	Key Features that Make Java More Secure than Other Languages.	579.



Preface

∞ Essential Java Skills—Made Easy! ∞

Learn the all basics and advanced features of Java programming in no time from Bestseller Java Programming Author Harry. H. Chaudhary (More than **1,67,000 Books Sold !**). This Java Guide, starts with the basics and Leads to Advance features of Java in detail with **thousands of Java Codes**, I promise this book will make you expert level champion of java. Anyone can learn java through this book at expert level.

Engineering Students and fresh developers can also use this book. This book covers common core syllabus for *all* Computer Science Professional Degrees If you are really serious then go ahead and make your day with this ultimate java book.

The main objective of this java book is not to give you just Java Programming Knowledge, I have followed a pattern of improving the question solution of thousands of Codes with clear theory explanations with different Java complexities for each java topic problem, and you will find multiple solutions for complex java problems.

What Special –

In this book I covered and explained several topics of latest Java 8 Features in detail for Developers & Fresher's, Topics Like— **Lambdas**. || **Java 8 Functional interface**, || **Stream and Time API in Java 8**.

*I*f you've read this book, you know what to expect a visually rich format designed for the way your brain works. If you haven't, you're in for a treat. You'll see why people say it's unlike any other Java book you've ever read.

Learning a new language is no easy task especially when it's an Object oriented programming language like Java. You might think the problem is your brain. It seems to have a mind of its own, a mind that doesn't always want to take in the dry, **technical stuff you're forced to study**. The fact is your brain craves novelty.

It's constantly searching, scanning, waiting for something unusual to happen. After all, that's the way it was built to help you stay alive. It takes all the routine, ordinary, dull stuff and filters it to the background so it won't interfere with your brain's real work—recording things that matter. How does your brain know what matters?

This Java book doesn't require previous programming experience. However, if you come from a C or C++ programming background, then you will be able to learn faster.

For this reason, this java book presents a quick detailed overview of several key features of Java. The material described here will give you a foothold that will allow you to write and understand simple & typical programs. Most of the topics discussed will be examined in greater detail in upcoming chapters with thousands of live java code examples.

As we know in the past few years document the following fact: The Web has irrevocably recast the face of computing and programmers unwilling to master its environment will be left behind. The preceding is a strong statement. It is also true.

More and more, applications must interface to the Web. It no longer matters much what the application is, near universal Web access is dragging, pushing, and coaxing programmers to program for the online world, and Java is the language that many will use to do it. Frankly, fluency in Java is no longer an option for the professional programmer, it is a requirement. This book will help you acquire it.





CHAPTER

∞ 1 ∞

(Overview of Java)

Introduction-

Java is a powerful object oriented programming language developed by Sun Microsystems Inc. in 1991. Java was developed for consumer electronic devices but later it was shifted towards Internet. Now Java has become the widely used programming language for the Internet. Java is a platform neutral language (Machine Independent). Program developed by Java can run on any hardware or on any operating system in this world.

Sun Microsystems (Oracle) formally announced Java at a major conference in May 1995. Ordinarily, an event like this would not have generated much attention. However, Java generated immediate interest in the business community because of the phenomenal interest in the World WideWeb.

Java is now used to create Web pages with dynamic and interactive content, to develop large-scale enterprise applications, to enhance the functionality of World Wide

Need for Java-

Java was developed due to the need for a platform neutral language that could be used to create software to be embedded in various consumer electronic devices, such as microwave ovens and remote controls. The program written in C and C++ are compiled for a particular piece of hardware and software and that program will not run on any other hardware or software. So we need C/C++ compilers one for each type of hardware to compile a single program. But compilers are expensive and time-consuming to create. So there is a need for platform neutral language. So that program compiled from that compiler can run on any hardware. This need led to the creation of Java.

Java Class Libraries-

Java programs consist of pieces called classes. Classes consist of pieces called methods that perform tasks and return information when they complete their tasks. You can program each piece you may need to form a Java program. However, most Java programmers take advantage of rich collections of existing classes in Java class libraries. The class libraries are also known as the Java APIs (Application Programming Interfaces).

Thus, there are really two pieces to learning the Java “world.” The first is learning the Java language itself so that you can program your own classes; the second is learning how to use the classes in the extensive Java class libraries.

Throughout the book, we discuss many library classes. Class libraries are provided primarily by compiler vendors, but many class libraries are supplied by independent software vendors (ISVs). Also, many class libraries are available from the Internet and World Wide Web as freeware or shareware. You can download free ware products and use them for free subject to any restrictions specified by the copyright owner.

Basics of a Typical Java Environment-

Java systems generally consist of several parts: An environment, the language, the Java Applications Programming Interface (API) and various class libraries. The following discussion explains a typical Java program development environment, Java programs normally go through five phases to be executed. These are: edit, compile, load, verify and execute. The descriptions that follow use the standard Java SE 7 Development Kit (JDK 7), which is available from Oracle.

If you are using a different Java development environment, then you may need to follow a different procedure for compiling and executing Java programs. In this case, consult your compiler’s documentation for details.

Note: If you are not using UNIX/Linux, Windows 95/98/ME or Windows NT/2000, refer to the manuals for your system’s Java environment or ask your instructor how to accomplish these tasks in your environment (which will probably be similar to the environment, Phase 1 consists of editing a file.

This is accomplished with an editor program (normally known as an editor). The programmer types a Java program, using the editor, and makes corrections, if necessary. When the programmer specifies that the file in the editor should be saved, the program is stored on a secondary storage device, such as a disk. Java program file names end with the .java extension.

Two editors widely used on UNIX/Linux systems are vi and emacs. On Windows 95/98/ME and Windows NT/2000, simple edit programs like the DOS Edit command and the Windows Notepad will suffice.

Java integrated development environments (IDEs), such as Forte for Java Community Edition, NetBeans, Borland's JBuilder, Symantec's Visual Cafe and IBM's Visual Age have built in editors that are integrated into the programming environment.

We assume the reader knows how to edit a file. Languages such as Java are object-oriented—programming in such a language is called object-oriented programming (OOP) and allows designers to implement the object oriented design as a working system. Languages such as C, on the other hand, are procedural programming languages, so programming tends to be action-oriented.

In C, the unit of programming is the function. In Java, the unit of programming is the class from which objects are eventually instantiated (a fancy term for “created”). Java classes contain methods (that implement class behaviors) and attributes (that implement class data).

C programmers concentrate on writing functions. Groups of actions that perform some common task are formed into functions, and functions are grouped to form programs. Data are certainly important in C, but the view is that data exist primarily in support of the actions that functions perform. The verbs in a system specification help the C programmer determine the set of functions needed to implement that system.

Java programmers concentrate on creating their own user-defined types called classes and components. Each class contains data and the set of functions that manipulate that data. The data components of a Java class are called attributes.

The function components of a Java class are called methods. Just as an instance of a built-in type such as int is called a variable, an instance of a user-defined type (i.e., a class) is called an object. The programmer uses built-in types as the “building blocks” for constructing user-defined types.

The focus in Java is on classes (out of which we make objects) rather than on functions. The nouns in a system specification help the Java programmer determine the set of classes from which objects will be created that will work together to implement the system.

Classes are to objects as blueprints are to houses. We can build many houses from one blueprint, and we can instantiate many objects from one class. Classes can also have relationships with other classes.

For example, in an object-oriented design of a bank, the “bank teller” class needs to relate to the “customer” class. These relationships are called associations. We will see that, when software is packaged as classes, these classes can be reused in future software systems. Groups of related classes are often packaged as reusable components.

Each new class you create will have the potential to become a valuable software asset that you and other programmers can use to speed and enhance the quality of future software-development efforts—an exciting possibility.

Relation of Java with C, C++, & C#

From C Java derives its syntax and from C++ it derives object oriented features. It is not an enhanced version of C++. Java is neither upwardly nor downwardly compatible with C++. One important thing that I want to tell you is that Java language was not designed to replace C++ and C#. Another language developed by Microsoft to support the .NET Framework, C# is closely related to Java because both share C++ and C style syntax, support distributed programming, and utilize the same object model.

Primary Objective of Java is to achieve -

1. Security: -

There is no threat of virus infection when we use Java compatible Web Browser. Also there is no threat of malicious programs that can gather private information, such as credit card numbers, bank account balances and passwords from local machine.

Java provides a firewall between a networked application and our computer.

2. Portability: -

Java programs are portable from one computer to another computer running different types of operating systems and having different hardware.

Java Bytecode -

The output of a Java compiler is bytecode not the machine code (".class" file). Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called as JVM (Java Virtual Machine).

JVM is the interpreter which interprets the bytecode. Compiled program runs faster but still Java uses interpreter to achieve portability so Java programs runs a little slower. Now a program compiled through a Java compiler can run in any environment but JVM needs to be implemented for each platform. Java programs are interpreted. This also helps to make it secure because the execution of every Java program is under the control of JVM.

JIT (Just In Time):-

JIT is a translator used by JVM to translate bytecode into actual machine code. It does not translate entire bytecodes rather it translates piece by piece on demand basis.

Various Versions of Java:-

Java 1

JDK 1.0

JDK 1.1

Java 2

JDK 1.2

JDK 1.3

JDK 1.4

JDK 1.5 or JDK 5

JDK 1.6 or JDK 6

Java 1.7 or SE 7

Java SE 8 (Java 8 , April 2014)

Note1- JDK (Java Development Kit)

Note2- Many features of old Java versions are deprecated by new versions of Java but still we can use them.

Type of applications Java can develop:-

1. **Standalone Applications-** A standalone application is a program that runs on our local computer under the operating system of that computer just like a C or a C++ program.
2. **Applets-** An applet is a small program which travel across the Internet and executed by a Java-Compatible web browser, such as Internet Explorer or Netscape Navigator, on the client machine.

An applet is actually a tiny Java program, dynamically downloaded across the network. Applet programs are stored on a web server and they travels to client machine on request from the client machine.

An applet cannot be executed like standalone application. Applet can be executed only by embedding it into an HTML page like a sound file or a image file or a video clip.

Now this HTML page which has applet embedded into it can be run after downloading such HTML page by a web browser on a local machine. An applet is a program that can react to user input and can change dynamically. It does not run the same animation or sound over and over.

3. **Web Applications-** These are the programs which run on Web Server. When we access a web site by specifying the URL (Universal Resource Locator) in a web browser then the web browser sends a request to the web server for a particular Web site. After receiving this request server runs a program and this program is called as Web Application. We use Java Servlets and JSP (Java Server Pages) to write such programs.

These programs run on the server and then send the result/response to the client. JSP pages can be thought of as a combination of HTML and Java Code. The Web Server converts JSP pages into Java Servlets before execution.

When a client request for a particular URL and the URL corresponds to an HTML page the web server simply returns the HTML page to the client, which then displays it. If the URL corresponds to the servlet or JSP then it is executed on the Server and the result/response is returned to the client, which is then displayed by the client.

4. **Distributed Applications-** Java application is divided into small programs which can run on separate machines. The objects used in these programs can communicate with each other. These applications are known as Distributed Applications. This allowed objects on two different computers to execute procedure remotely. For this RMI (Remote Method Invocation) is used.

Characteristics of Java:-

1. **Simple-** The syntax of Java is almost similar to C and C++ so that a programmer is familiar with C/C++ does not have to learn the syntax from scratch. But many features of C/C++, which are either complex or result in ambiguity have been removed in Java.

1. Java does not support multiple inheritance, as the concept is a bit complex and may result in ambiguity.
 2. Java does not support global variables, which also lead to many bugs in C/C++ programs.
 3. Java does not use pointers and does not allow pointer arithmetic, which is cause of most of the bugs in C/C++ programs due to inherent complexity.
 4. Java does not support operator overloading as it may lead to confusion.
 5. There is no concept of garbage value in Java. We have to initialize variables before use.
2. **Secure-** Java programs run within the JVM (Java Virtual Machine) and they are inaccessible to other parts. This greatly improves the security. A Java program rarely hangs due to this feature. It is quite unlike C/C++ programs, which hang frequently. Java's security model has three primary components:

1. ***Class loader.***
2. ***Bytecode Verifier.***
3. ***Security Manager.***

Java uses different class loaders to load class files (executable files) from local machine and remote machines. The classes loaded from remote machines like Applet classes are not allowed to read or write files on the local machine. This prevents a malicious program from damaging the local file system. Bytecode verifier verifies the bytecode as soon as class loader completes its work. It ensures that bytecode is valid Java code. It almost eliminates the possibility of Java program doing some malicious activity like accessing the memory outside the JVM. The Security Manager controls many critical operations like file deletion, creation of threads etc. These operations are allowed only if the Java programs have sufficient permissions otherwise Security Manager does not allow the operations and generates Security Exception.

3. **Portable-** Java programs are platform independent. They follow the policy of write-once-run-anywhere. A Java program written for Windows Platform can run on any other platform (Unix, Linux, Sun Solaris etc.) simply by copying the bytecode (".class" files). There is no need to copy the source code and compile it again as in case of a C/C++ program. This feature has made the Java a powerful language. We can run bytecode on any machine provided that the machine has the JVM. JVM is itself is platform dependent but it makes the Java code platform independent. It is actually JVM which converts the bytecode into machine code and executes them.

So we can say that Java is a portable language. One more feature which makes Java highly portable is that primitive data types are of fixed length irrespective of the platform. For example an int will always be 4 bytes in Java. This is unlike C/C++ where size of int can be 2 bytes on some machines and 4 bytes on other machines.

4. **Object Oriented-** Java is almost pure object-oriented language but it supports primitive data types like byte, short, int, long, float, double, char, boolean for the performance reasons.
5. **Robust:-** Most programs fail one of the two reasons:
 1. *Memory Management.*
 2. *Exceptional conditions at run time.*

While designing the language one of the aim was to ensure that Java programs are as robust as possible i.e. they should rarely fail. So due importance was given to the above two factors in the Java.

In Java memory allocation and de-allocation is handled in the language itself, which eliminates many problems caused due to dynamic memory management features in C/C++. Java also supports object oriented exceptional handling features to handle exceptional conditions, which occur at run-time. This allows a Java program to recover and continue execution even after an exceptional condition occurs.

6. **Multithreaded:-** Java was designed to meet the real world requirement of creating interactive, networked programs. Java provides support for writing multi-threaded programs to achieve this. This allows the programmer to write programs that can do many things concurrently.

For example a GUI (Graphical User Interface) based application might be listening to user events and taking appropriate action, a separate thread might be doing printing and a separate thread might be downloading a file from some machine across the network, all of this being done concurrently. This results in better performance and better CPU utilization.

It is possible to write multi-threaded programs in other languages also but it is achieved only by making use of System calls while in case of Java it can be achieved by using features of the language itself.

7. **Architecture-neutral-** One of the main problems facing programmers is that

no guarantee exists that if we write a program today, it will run tomorrow even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. But the goal of Java programs is “write once run anywhere”.

8. **Interpreted and High Performance-** Java programs are interpreted but still they run fast as compared to other interpreters.
9. **Distributed-** Java is designed for distributed environment of the Internet. Java has built-in support for various TCP/IP based protocols for this purpose. In fact accessing a resource using a URL is similar to accessing a file on the local machine. Java also has features for Remote Method Invocation, which is somewhat similar to Remote Procedure Calls (RPC). This allows objects on different computers to execute procedures remotely. Java has built-in API's for this purpose called as RMI.
10. **Dynamic-** Every Java class is a separate unit of execution. A class is loaded at the run time only when it is needed. Default mechanism for binding methods in Java is also dynamic (run-time binding).

Running a Standalone Java Application:-

When a C or C++ program is compiled, it is directly translated into machine code of a particular processor or a particular platform. But running a Java program is a two-step process. In Java translation from source code to the executable code is achieved using two translators:

1. **Java Compiler** - First of all Java program is compiled into bytecode. Bytecode are just like machine code but not for a particular processor or platform. Bytecodes can not be directly executed.
2. **Java Interpreter (JVM)** - Java interpreter by using JIT translates the bytecode into actual machine code of a particular platform.

Note 1- C or C++ programs are compiled only once but Java bytecodes are translated every time we execute Java programs. So Java programs run a little slower as compared to C/ C++ programs.

Note 2- To run a Java program we need a Text Editor (Notepad or Edit), JDK (Java Development Kit), and JVM (already installed in many operating system). We can also use Eclipse or JCreator in place of Notepad.

Installing Java 1.7 on win xp-

Double Click on My Computer Double Click on CD Drive Double Click on JDK1.7 Java Installation File Press Next, Next,

To set the Path -

If we set the path of Java folder then we can run Java programs from anywhere otherwise we have to run our java programs from the bin folder of Java.

Right Click on My Computer Properties Advanced Environment Variables Click on Path and then click on Edit Click on the variable value Move the cursor at the end of this line by pressing END Now type “c:\program files\Java\jdk1.7.0_01\bin; and then click on OK, OK, OK.

Creating a folder for Java programs:-

Double Click on My Computer Double Click on C: Drive Right Click New Folder Type JAVAPRG Press Enter and close the My Computer.

Steps for running Java Program on a command Prompt:-

Step 1:- Go To Command Prompt

Start Run Type “cmd” and click on OK.

Step 2:- Type “CD \ JAVAPRG” and press Enter

Step 3:- Type “Edit Sum.Java” and press Enter

Step 4:- Type the Java Program and click on File Save and then File Exit

Step 5:- Type “javac Sum.Java” and press enter to compile. This will create “Sum.class” file. This file is known as bytecode.

Step 6:- Type “java Sum” and press enter to run. This will invoke the JVM.

Step 7:- Type “Exit” and then press enter to exit from command prompt.

Note - We can also run Java programs through Notepad using above steps but instead of using Edit command we will use Notepad. After typing the Java Program save it to the folder “C:\JAVAPRG” after choosing “All files” in save as dialog box. To compile and run we have to follow above steps.

Java Program Syntax:-

```
class <class name>
{
public static void main(String args[])    or            (String [] args)
{
- - - - -
```

```
    }  
}
```

First Java Program - Program to add two numbers (Sum.Java)

```
1.    class Sum  
2.    {  
3.        public static void main(String args[])  
4.        {  
5.            int a,b,sum;  
6.            a = 5;  
7.            b = 6;  
8.            sum = a + b;  
9.            System.out.print("Sum is " + sum); //Output  
10.        }  
11.    }
```

Output: Sum is 11.

Example 1.2 Program to swap two numbers (Swap.Java)

```
1.    class Swap  
2.    {  
3.        public static void main(String args[])  
4.        {  
5.            int a=5,b=7;  
6.            t = a;  
7.            a = b;  
8.            b = t;  
9.            System.out.print("After Swap Values are " + a + "," + b);  
10.        }  
11.    }
```

Output: After Swap Values are 7,5

Comments-

There are 3 types of comments.

- Single line comment //
- Multi line comment /* */
- Documented comment /** */

Every Java program must contain one class. In Java main() method can be defined only inside a class.

A source file may have any number of class and or interface definitions but there can be at the most one public class or interface.

A source file may or may not contain a class having main() method but a standalone Java program always starts its execution from main() method. So the class from which we want to start the execution must have the main () method defined in it.

The main() method must always have the following signature:

- public static void main(String args[])

Note -

Method prototype is referred to as signature in Java.

The keyword public indicates that the method can be accessed from anywhere. The main() method must be declared public as it is called by the code, which is part of the JVM and is outside the class containing the main() method.

The keyword static indicates that main() is a class level method and can be called without creating any object it is must as no object exists before main() is called and any object creation occurs only inside the main() method.

The void keyword preceding main() method indicates that it does not return

any value.

The main() method takes one argument, which is the array of Strings. Each element of the array represents one command line argument.

System is a predefined class that provides access to the system, and out is the output stream that is connected to the console. print() method can be used to display any type of information.

Structure of a Java Program –

All Java source files must end with the extension “.Java”. Java is a case sensitive language. Note that in the above program first letter of classes Sum, String and System is written in uppercase. It is better to save the Java program with the file name which is same as of class name, but it is not compulsory. But if class is declared as public then it is compulsory to save the class in a file whose name is same as of class name. A class can contain only one public class but may contain as many non-public classes. If a file contains more than one class then after compilations each individual class is put into its own output file named after the class and using .class extension.

Documentation Section -The documentation section contains a set of comment lines giving the name of the program, the author and other details, which the programmer would like to refer to at a later stage. Comments explain why and what of classes and how of algorithms. In addition to single, multi line comment Java also uses a third style of comment `/**...*/` known as documentation comment.

Package Statement -A source file may have at the most one package statement. If a package statement is present it must be the first statement in the Java program. Only comments may appear before the package statement.

Import Statements - A source file may have zero or more import statements. Import statements are just like `#include` statement in C/C++. If present all the import statements must come after the package statement and before the class/interface definition.

Interface Statements -An Interface is like a class but includes a group of method declarations. This is also an optional section and is used only when we wish to implement the multiple inheritance features in the program.

Class definition not containing main method -A Java program may contain multiple class definitions. Classes are the primary and essential elements of a Java Program. These classes are used to map the objects of real-world problems.

Class definition containing main method -Since every Java stand-alone

program requires a main method as its starting point, this class is the essential part of a Java program. A simple Java program may contain only this part. The main method creates objects of various classes and establishes communications between them. On reaching the end of main, the program terminates and the control passes back to the operating system.

Garbage Collection -

Memory allocation for the Java objects is completely dynamic but Java does not have support for pointer arithmetic like C/C++. Whenever we run a Java program, JVM also runs another program (thread) called Garbage Collector in the background. Garbage Collector keeps check on the Java objects.

Whenever a Java object is not being used it is collected by the Garbage Collector i.e. the memory allocated for the object is added to the pool/heap of free memory and can be reused. This simplifies the task of the programmer to a large extent. This also eliminates lots of bugs caused due to improper use of pointer arithmetic and memory management features like freeing memory explicitly.

Multiple choice Questions:

1. Which of the following are valid definitions of an application's main() method?

- (a) `public static void main();`
- (b) `public static void main(String args);`
- (c) `public static void main(String args[]);`
- (d) `public static void main(Graphics g);`
- (e) `public static boolean main(String args[]);`

2. Which organization developed the Java language?

- (a) Microsoft
- (b) IBM
- (c) Sun Microsystems Inc.
- (d) AT&T

3. Java belongs to which of the following language categories?

- (a) Object-Oriented
- (b) Procedural
- (c) Both (a) & (b)
- (d) None of the above

4. Which of the following is not a correct statement?

- (a) Garbage collection in Java is automatic.
- (b) Java provides support for distribution applications.
- (c) Java is more efficient as compared to C/C++.
- (d) Java is more secure as compared to C/C++.

5. Which of the following is a correct statement?

- (a) Java is a platform independent language?
- (b) Java is an Object-Oriented language?
- (c) Java supports multi-threading.
- (d) All of the above.

Answers: 1. (c) 2. (c) 3. (a) 4. (c) 5. (d)

Theory Questions:

1. Explain working of Java Virtual Machine (JVM).
2. What are the differences between C++ and Java?
3. Difference between “APPLET” and “APPLICATION”.
4. Disadvantages of Java.
5. How does garbage collection work?
6. What is BYTE Code?
7. Is java a fully object oriented programming or not? if not why?
8. What are the different types of Java Applications / Programs?
9. What is the extension of a Java executable file?
10. What is the extension of a Java source / program file?
11. Which command is used to compile a Java Program?
12. Which organization developed the Java language?
13. What makes Java platform independent?
14. Why Java is more secure language as compared to C/C++?
15. Why Java is more robust language as compared to C language?
16. What do we understand by distributed application?
17. Describes any four features of Java.
18. Describes the steps needed to compile and run a Java program.
19. What are the main reasons for the popularity of Java?

20. Why many features of C/C++ have been removed in Java? Name some of these features.
21. Name three types of comments in Java.
22. List any 10 major differences between C and Java.
23. Describe the structure of a typical Java program.

Programming Exercise:

1. Write a program to calculate average of two integer numbers.
2. Write a program to swap 2 numbers without using 3rd variable.
3. Write a program to convert hours, minutes and seconds into total seconds.
4. Write a program to calculate the area and circumference of a given circle.





CHAPTER

∞ 2 ∞

(Java Language)

Tokens-

The smallest individual units are known as tokens such as keywords, identifiers, constants, strings & Operators.

A. Keywords are the reserved names of a language and cannot be used as names of variables, functions etc.

B. Identifier refers to the names of variable, arrays, functions, classes, Interfaces etc.

C. Constants/Literals refer to fixed values that we cannot change in a program.

D. Operators are special symbols which operate on variable & constants, and form an expression.

E. Separators are the special characters used to separate statements such as “() {} [] ; , .”

A. Keywords-

abstract	continue	goto	package	
synchronized				
assert	default	if	private	
this				
boolean	do	implements		
protected	throw			
break	double	import		
public	throws			
byte	else	instanceof	return	
transient				
case	extends	int	short	try
catch	final	interface	static	
void				
char	finally	long		
strictfp	volatile			
class	float	native	super	
while				
const	for	new	switch	
enum				

Note1- There are 50 reserved keywords in Java. The keyword `const` and `goto` are reserved but not used. The `assert` keyword was added by Java 2 version 1.4. The `enum` keyword was added by Java2 version 1.5.

Note2- In addition to keywords, Java reserves the following literals **true**, **false**, and **null**. We can't use these reserved names as Identifiers (Variable or Class or Interface names).

B. Identifiers:-

An identifier is a word used in a program to name a variable, method (function), class, interface, package etc. Java identifiers are case sensitive.

Identifier naming rules (we must follow):-

1. *A Java identifier must begin with a letter, dollar sign or underscore. It can't begin with a number.*
2. *The subsequent characters may be digits also.*
3. *There is no restriction on the length of identifiers.*
4. *Java allows Unicode characters in identifiers. We can use `€`, `£`, `¥`, etc. in identifiers as they are treated as letters in Unicode (ASCII is replaced by Unicode in Java).*

Identifier naming rules (we should follow)-

1. Names of all the classes and interfaces start with a leading uppercase letter and each subsequent word also starts with a leading uppercase letter. Rest of the letters must be in lower case. (Example StudentTest).
2. Names of all the public data members and methods start with a leading lowercase character. When more than one word are used in a name, the second and subsequent words start with a leading uppercase letter. (Example netAnnualProfit).
3. Names of variables that represent constant values use all uppercase letters and underscores between words. (Example PI).

C. Constants/Literals:-

A Literal represents a constant value which we can't change. A literal can't appear on the left side of an assignment operator. A literal is a value specified in the source code it is not determined at run time.

Type of Literals-

1. 5, -5, 0x5A, 012, 5l, 5L are **Integer Literals** in which 5,-5 are decimal, 0x5A is a hexa decimal, 012 is a octal, l or L is for long.
2. 5.23, -5.23, 5.4f, 5.4F, 5.4d, 5.4D, 1.2E-03 are **Floating Point Literals** where f or F is for single precision(float), d or D is for double precision, 5.23 is a Standard notation and 1.2E-03 is in exponent notation (Scientific notation) where 1.2 is mantissa and -03 is exponent.

3. 'a', 'A', '\n', '\141', '\u0061' are **Character Literals** in which '\n' is an escape sequence, '\141' is octal code for character 'a' and '\u0061' is hexa-decimal code for character 'a'.
4. "matrix" is a **String Literal**. String is a group of characters. In Java Strings there is no line-continuation escape sequence as there in other languages. So Strings must begin and end on the same line. In Java strings are of object type. "\nThis is in quotes\n".
5. True, false are **Boolean Literals**. True and false are not equal to 1 or 0 in Java. We can't convert a Boolean value to integer and vice-versa.

D. Operators-

Java provides a rich set of operators. Operators combine constants, variables and sub-expressions to form expressions. Most of the operators in Java behave like C/C++ but there are few differences, which are covered here.

Operators can be classified as:

Arithmetic Operators (+,-,*,/,%)

These operators are same as in C/C++.

% operator can work on floating numbers also.

When binary operator is applied on two operands of different types then operand of lower type gets converted to the higher type before the evaluation and the type of the result will be same as that of operand of higher type.

Division or modulus by zero result in ArithmeticException (run time Error) in case of int but not in case of floating numbers.

When we apply arithmetic operators on int and the result is outside the range of int then extra high order bits will be truncated and this new value will be

assigned to the variable receiving their result but in floating type in case of overflow it will result in Infinity or -Infinity and in case of underflow it will result in 0.

Increment and Decrement Operators (++, —)

These operators are same as in C/C++.

```
int a=5,b;
b = ++a*++a;
System.out.print(b);           //42
a = 5;
System.out.print(++a*++a);     //42
```

Relational Operators (<, >, <=, >=, instanceof)

These operators are same as in C/C++ but <.<=,>.>= cannot be applied on boolean types and reference types.

instanceof operator:-

The instanceof operator is used to test the class of an object. The instance of operator has the general form:

if(object instanceof type)

Here, object is an instance of a class, and type is a class type. If object is an instance of the specified type or instance of any sub-class of the specified type, then the instance of operator return true otherwise its result is false.

Assignment Operators (=, +=, -=, *=, /=, %=)

These operators are same as in C/C++.

Equality Operators (=, !=)

These operators are same as in C/C++.

Logical Operators (&&, ||, !)

These operators are same as in C/C++.

Conditional Operator (? :)

This operator is same as in C/C++.

Bitwise Operators (&, |, ^, ~, <<, >>, >>>)

These operators are same as in C/C++ except shift operators. Shift operators works only on integer type.

Left Shift (<<) Operator

To obtain the result of << operator, the bits in the left hand side operand are shifted to the left as specified by the right hand operand and the empty bit positions to the right are filled with zero. Left shifting by 1 is equivalent to multiplication by 2. It is possible that sign of the result may differ from the sign of the left hand side operand. This may happen because the sign depends on the left-most bit, which can change from 0 to 1 or 1 to 0 hence the change in sign.

Example $b = a \ll 2$; To obtain the value of b, shift the bits in a by 2 positions to the left and fill the 2 right bits with zero.

Right Shift (>>) Operator (signed)

To obtain the result of >> operator, the bits in the left hand operand are shifted to the right as specified by the right hand operand and the empty bit positions to the left are filled with sign bit. Right shifting by 1 is equivalent to division by 2. This operator never changes the sign of the result i.e. it will be same as the sign of the left hand operand.

Example $b = a \gg 2$; To obtain the value of b, shift the bits in a by 2 positions to the right and fill the 2 right bits with sign (0 if a is positive or 1 if a is negative.).

Right Shift with zero fill (>>>) Operator (unsigned)

To obtain the result of >>> operator, the bits in the left hand side operand are shifted to the right as specified by the right hand operand and the empty bit positions to the left are filled with 0. Right shifting by 1 is equivalent to division by 2. If shifting takes place then result will always be positive, as the rightmost bit would become zero.

Example $b = a \ggg 2$; To obtain the value of b, shift the bits in a by 2 positions to the right and fill the 2 right bits with 0.



Precedence of Operators:-

Rank	Operators	Description	Associativity
1	()	Function call	Left to Right
	[]	Subscript	
	.	Direct member	
2	++	Increment	Right to Left
	--	Decrement	

E. Separators:-

	~	Bitwise unary NOT	
	!	Logical unary NOT	
3	*	Multiplication	Left to Right
	/	Division	
	%	Modulus	
4	+	Addition	Left to Right
	-	Subtraction	
5	>>	Bitwise Right Shift	Left to Right
	>>>		
	<<	Bitwise Left Shift	
		Bitwise Right Shift Zero Fill	
6	>	Greater than	Left to Right
	>=	Greater than or equal to	
	<	Less than	
	<=	Less than or equal to	
	instanceof		
7	==	Equal To	Left to Right
	!=	Not Equal To	
8	&	Bitwise AND	Left to Right
9	^	Bitwise XOR	Left to Right
10		Bitwise OR	Left to Right
11	&&	Logical AND, circuit AND	Left to Right
12		Logical OR, Short circuit OR	Left to Right

13	?:	Conditional operator (Ternary)	Left to Right
14	=	Assignment	Right To Left
	+=	Addition assignment	
	-=	Subtraction assignment	
	*=	Multiplication assignment	
	/=	Division assignment	
	%=	Modulus assignment	
	&=	Bitwise And assignment	
	=	Bitwise Or assignment	
	^=	Bitwise XOR assignment	
	>>=	Bitwise Right Shift assignment	
	<<=	Bitwise Left Shift assignment	
	>>>=	Bitwise Right Shift Zero Fill assignment	

Symbol	Name	Purpose
()	Parenthesis	Used to contain list of parameters in method definition and invocation (calling). Also used for defining precedence in expressions, containing expressions in control statements, and in typecasting.

{ }	Curly Braces	Used to initialize arrays, Also used to define a block of statements, for classes, interfaces, methods and local scope.
[]	Square Brackets	To declare an array.
;	Semicolon	To terminate a statement.
,	Comma	Separates consecutive identifiers in a variable declaration, Also used to chain statements together inside a for statement.
.	Period	Used to separate package names from sub-packages and classes. Also used to separate a variable or method from a reference variable.



Data Types- Data types in Java can be broadly classified in two categories:

1.Primitive Data Types/ Simple Data Types-

Java is an object oriented language, but the primitive data types are not objects. They are kept in Java for performance reason. They form the basis for all other types of data that we define in our Java programs. Java is a strongly typed language.

Numeric Data Types

1. **Non-Primitive Data Types/ Derived Data Types or Reference**

Data Types- Reference data types are also called derived data types as they are derived from the primitive data types.

Classes

1. Built-In/Library Classes
2. User-Defined Classes

Interfaces

3. Built-In/Library Classes
4. User-Defined Classes

Arrays - Arrays are also treated as objects in Java.

Java's Automatic Conversions will take place when-

1. The two types are compatible.
2. The destination type is larger than the source type.

Variables-

Variables are the name of the memory locations that can store values. A variable must be declared before we can store value in it. Or Variable is the place (Memory Location) inside the main memory where we can store our data or values.

There are three kinds of variables:

Local Variables:- Local variables are used inside blocks or methods. Local variables (also known as automatic variables) are not initialized by default. A local variable must be explicitly initialized before being used for the first time otherwise a compile time error "Variable might not have been initialized" will come.

Example

```
int a,b=5;
if(b>2)
{
    a = b;
}
System.out.print(a);
```

In the above program the value of a is not defined if the if condition is false, so the compiler will give an error “Variable a might not have been initialized”.

In Java we can not define a variable in the inner block with the same name as of outer block.

```
int a=5;
{
    int a=7;    // Error
}
```

Instance Variables:- Instance variable are used to define attributes or state of an object.

Class Variables:- Class variables are used to define attributes/state, which is common for all the objects of a class.

Library Methods of class Math-

public static double sin(double x);	Return the sine of the angle x in radians
public static double cos(double x);	Return the cosine of the angle x in radians

<code>public static double tan(double x);</code>	Return the tan of the angle x in radians
<code>public static double asin(double x);</code>	Return the angle x of sine
<code>public static double acos(double x);</code>	Return the angle x of cosine
<code>public static double atan(double x);</code>	Return the angle x of tan
<code>public static double toRadians(double x);</code>	Convert degrees x to radians
<code>public static double toDegrees(double x);</code>	Convert radians x to degrees
<code>public static double exp(double x);</code>	Return e raised to $x(e^x)$
<code>public static double log(double x);</code>	Return the natural logarithm of x
<code>public static double log10(double x);</code>	Return the natural logarithm of x base 10
<code>public static double sqrt(double x);</code>	Return the sqrt of x
<code>public static double ceil(double x);</code>	Return the smallest whole number greater than or equal to x (rounding up)
<code>public static double floor(double x);</code>	Return the largest whole number less than or equal to x (rounded down)
<code>public static double rint(double x);</code>	Return the rounded value of x
<code>public static double atan2(double x, double y);</code>	Return the angle whose tangent is x/y
<code>public static double pow(double x, double y);</code>	Return x raised to $y(x^y)$

<code>public static int round(float x);</code>	Return the rounded value of x in int
<code>public static long round(double x);</code>	Return the rounded value of x in long
<code>public static double random();</code>	Returns a random number between 0 to 0.9999999999999999999
<code>public static int abs(int x);</code>	Return the absolute value of x.
<code>public static long abs(long x);</code>	Return the absolute value of x.
<code>public static float abs(float x);</code>	Return the absolute value of x.
<code>public static double abs(double x);</code>	Return the absolute value of x.
<code>public static int max(int x, int y);</code>	Return the maximum of x & y
<code>public static long max(long x, long y);</code>	Return the maximum of x & y
<code>public static float max(float x, float y);</code>	Return the maximum of x & y
<code>public static double max(double x, double y);</code>	Return the maximum of x & y
<code>public static int min(int x, int y);</code>	Return the minimum of x & y
<code>public static long min(long x, long y);</code>	Return the minimum of x & y
<code>public static float min(float x, float y);</code>	Return the minimum of x & y
<code>public static double min(double x, double y);</code>	Return the minimum of x & y

public static double sinh(double x);	Returns the sin hyperbolic of angle x
public static double cosh(double y);	Returns the cos hyperbolic of angle x
public static double tanh(double x);	Returns the tan hyperbolic of angle x

Note- To use above methods we have to prefix Math class name.



Examples:

- ```
double theta1=120.0;
double theta2=1.312;
System.out.println(theta1+" degree is "+ Math.toRadians(theta1) +" radians.");
System.out.println(theta2 +" radians is "+Math.toDegrees(theta2) +" degrees");
```

### Output:

```
120.0 degree is 2.0943951023931953 radians.
1.312 radians is 75.17206272116401 degrees.
```

- ```
double number,root;
number =25.0;
root=0.0;
root=Math.sqrt(number);
```

```
System.out.println("Sqrt of number " + number + " is " + root);
```

Output:

Sqrt of number 25.0 is 5.0

```
3.      double x=3,y=3,z=0;
z=Math.pow(x,y);
System.out.println("Value of z :"+z);
```

Output:

Value of z :27.0

```
4.      double a=3.0,b=4.0; //dynamic initialization
double c=(Math.sqrt(a*a+b*b));
System.out.println("Hypotenuse is "+c);
```

Output:

Hypotenuse is 5.0



Multiple Choice Questions:

1. Which of the following are Java keywords?

- (a) array
- (b) boolean
- (c) Integer
- (d) protect
- (e) super

2. Which identifier is invalid?

- (a) `_xpoints`
- (b) `r2d2`
- (c) `bBb$`
- (d) `set-flow`
- (e) `thisisCrazy`

3. An integer, x has a binary value (using 1 byte) of 10011100. What is the binary value of z after these statements:

```
int y = 1 << 7;
```

```
int z = x & y;
```

- (a) 1000 0001
- (b) 1000 0000
- (c) 0000 0001
- (d) 1001 1101
- (e) 1001 1100

4. Which statements are accurate:

- (a) `>>` performs signed shift while `>>>` performs an unsigned shift.
- (b) `>>>` performs a signed shift while `>>` performs an unsigned shift.
- (c) `<<` performs a signed shift while `<<<` performs an insigned shift.
- (d) `<<<` performs a signed shift while `<<` performs an unsigned shift.

5. Consider the two statements:

1. `boolean passingScore = false && grade == 70;`

2. `boolean passingScore = false & grade == 70;`

The expression

```
grade == 70
```

is evaluated:

- (a) in both 1 and 2
- (b) in neither 1 nor 2
- (c) in 1 but not 2
- (d) in 2 but not 1
- (e) invalid because false should be FALSE

6. Given the variable declarations below:

```
byte myByte;
int myInt;
long myLong;
char myChar;
float myFloat;
double myDouble;
```

Which one of the following assignments would need an explicit cast?

- (a) myInt = myByte;
- (b) myInt = myLong;
- (c) myByte = 3;
- (d) myInt = myChar;
- (e) myFloat = myDouble;
- (f) myFloat = 3;
- (g) myDouble = 3.0;

7. Which of the following Java reserved word has no use as of now.

- (a) true
- (b) false
- (c) goto
- (d) null

8. What is the size of data type **int** in Java?

- (a) 2 bytes
- (b) 4 bytes
- (c) not defined
- (d) 2 bytes or 4 bytes

Answer's:

- | | | | | | | | |
|---|-------|---|-------|---|-----|---|-----|
| 1 | (b,e) | 2 | (d) | 3 | (b) | 4 | (a) |
| 5 | (d) | 6 | (b,e) | 7 | (c) | 8 | (b) |



Theory questions:

1. Why Java is not 100% pure object oriented language?
2. What are the primitive types in Java
3. Name four top-level elements that may appear in a Java source file.
4. What other statement(s) can appear before a package statement?
5. Name the three Java reserved words, which are used as literals.
6. Name two Java reserved words, which are not used in the language
7. Which primitive data-types of Java are not there in C?
8. Name three kinds of variables in Java.
9. Briefly describing rules for Java identifiers.
10. What is the purpose of comments in a Java program?
11. Name any four numeric data types along with their range.
12. How arrays in Java are different from C?
13. What are Java tokens? Give brief description of different types of Tokens.
14. What are Java literals? Explain with examples.
15. What are the different data types in Java? Describe in detail.

Programming Exercise:

1. Write a program which will show the use of all the bitwise operators.
2. Write a program which will show all explicit and implicit conversions.
3. Write a program which will solve some equations of different data types and display the result.
4. Write a program to proof that name of a inner scope variable can not be same as outer scope variable.
5. Write a program to calculate area of a circle. (use final to declare a constant).
6. Write a program to convert the char to Unicode.
7. Write a program to type conversion int to float.

8. Write a program to calculate modulus(%) of two float numbers.
9. Write a program to find the roots of a quadratic equation.



CHAPTER

∞ 3 ∞

(Control Statement)

Introduction-

Control statements are used to change the flow of execution. Java's control statements can be classified into three categories.

1. **Selection Statements (Decision Control Structure)** Selection statement allows the program to choose any one path from different set of paths of execution based upon the outcome of an expression or the state of a variable.
 1. if
 2. if else
 3. nested if else
 4. if-else-if
 5. switch

(a) Syntax of if

```
if(condition)
{
    statements;
}
```

(b) Syntax of if else

```
if(condition)
{
}
else
{
}
```

(c) Syntax of Nested if else

```
if(condition)
{
    if (condition)
    {
    }
    else
    {
    }
}
else
{
    if (condition)
    {
    }
    else
    {
    }
}
```

(d) Syntax of if-else-if Ladder

```
if(condition1)
{
}
else if(condition2)
{
}
else if(condition3)
{
}
else
{
}
```

(e) Syntax of switch case

```
switch(expression)
{
    case 1:
        statement 1 sequence;
```



```

        break;
    case 2:
        statement 2 sequence;
        braek;
    case 3:
        statement 3 sequence;
        break;
    —
    —
    case n:
        break;
    default :
        default statement sequence;
}

```

Example 3.1 Program to find max of two numbers (Max.Java)

```

2.     class Max
3.     {
4.         public static void main(String args[])
5.         {
6.             int a,b,max;
7.             a = 10;
8.             b = 20;
9.             if(a>b)
10.                 max = a;
11.             else
12.                 max = b;
13.             System.out.println("Max of two numbers is " + max); //Output
14.         }
15.     }

```

Output:

Max of two numbers is 20

2. **Iteration Statements (Loop Control Structure)** Looping is a process by which we can repeat a single statement or a group of statements n number of times.

1. **for**
2. **while**
3. **do while**

(a) **Syntax of for loop.**

```
for( initialize; condition; increment/decrement)
{
    statements;
}
```

(b) **Syntax of while loop.**

(c)

```
initialize;
while(condition)
{
    statements;
    increment/decrement;
}
```

(d) **Syntax of do while loop.**

```
initialize;
do
{
    statements;
    increment/ decrement;
} while(condition);
```

Note-

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control pass to the next line of code immediately following the loop.

Example 3.2

```
1.      class Loop1
2.      {
3.          public static void main(String args[])
4.          {
5.              int n=5;
6.              for(int i=1;i<=n;i++)
7.              {
8.                  for(int j=1;j<=i;j++)
9.                      System.out.print(j);
10.                     System.out.println();
11.             }
12.         }     }
```

Output:

```
1
12
123
1234
12345
```

3. **Jump Statements-** It allows our program to work in a non-linear fashion.

1. **break.**
2. **labeled break.**
3. **continue.**
4. **labeled continue.**
5. **return.**

Most of the statements have same syntax as the corresponding statements in C/C++. However there are some significant differences.

The conditional expressions used in the if, for, while and do statements must be valid boolean expressions i.e. their values should be either true or false. We

can not use 0 instead of false or a non-zero value instead of true that is valid in C/C++.

Java does not have any goto statement although goto is a reserved word in the language. Java provides labeled break and labeled continue statements which are often referred to as the civilized form of goto as they are supposed to be better substitutes of goto statement.

In Java, switch expression can be any integer expression except long i.e. the type of the switch expression can be byte, short, char or int. But the Java's int is of 4 bytes which is same as size of long in C/C++.

The case labels are constant expressions as in C/C++ but the values of the case labels must be in the range of the type of the switch expression otherwise the program will not compile.

The labeled break statement is used to terminate the block whose label is specified in the break statement. Unlike simple break statement, we can terminate any block. For example, it is possible to terminate the outermost loop from inside a deeply nested for loop. The break statement can also be used to terminate a simple block (i.e. the block need not be a loop or switch statement)

The labeled continue statement specifies the label of the enclosing loop to continue. The label need not correspond to the closest enclosing loop.

Jump Statement:-

```
(a)    break
        for(int i=1;i<=10;i++)
        {
            if(i == 5)
                break;           //terminate loop if i is 5
            System.out.print( i );
        }
```

Output: 1 2 3 4

(b) **labeled break**

```
boolean t =true;
    first :      {
                second: {
                    third: {
                        System.out.println("Before the break");
                        if(t) break second; //break out of second block
                        System.out.println("This will not execute");
                    }
                    System.out.println("This will not execute");
                }
                System.out.println("This is after second block");
            }
```

Output: Before the break
This is after second block

(c) **continue**

```
for(int i=0;i<10;i++)
{
    System.out.print( i + " ");
    if ( i % 2 == 0 )
        continue;
    System.out.println();
}
```

Note- This code use the % operator to check if i is even. if it is even then the loop will continue without printing a new line.

Output:

0 1
2 3

4 5

6 7

8 9

(d) **labeled continue**

```
outer : for(int i=0;i<5;i++)
    {
        for(int j=0;j<5;j++)
        {
            if ( j > i )
            {
                System.out.println();
                continue outer;
            }
            System.out.print("\t" + ( i * j ) );
        }
    }
```

Output:

```
0
0    1
0    2    4
0    3    6    9
0    4    8    12    16
```

(e) **return**

```
int sum(int a, int b)
{
    int c;
    c = a+b;
    return(c);
}
```

1. Given the variables defined below:

```
int one = 1;  
int two = 2;  
char initial = '2';  
boolean flag = true;
```

Which of the following are valid?

- (a) `if(one){}`
- (b) `if(one = two){}`
- (c) `If(one == two){}`
- (d) `if(flag){}`
- (e) `switch(one){}`
- (f) `switch(flag){}`
- (g) `switch(initial){}`

2. If `val = 1` in the code below:

```
switch( val )  
{  
  
    case 1: System.out.print( "P" );  
    case 2:  
    case 3: System.out.print( "Q" );  
        break;  
    case 4: System.out.print( "R" );  
    default: System.out.print( "S" );  
  
}
```

Which values would be printed?

- (a) P
- (b) Q
- (c) R
- (d) S

3. Assume that `val` has been defined as an positive int for the code below:

```
if( val > 4 )  
    System.out.println( "Test A" );  
else if( val > 9 )  
    System.out.println( "Test B" );  
else  
System.out.println( "Test C" );
```

Which values of val will result in “Test C” being printed:

- (a) val < 0
- (b) val between 0 and 4
- (c) val between 4 and 9
- (d) val > 9
- (e) val = 0
- (f) no values for val will be satisfactory

4. For the code:

```
m = 0;  
while( m++ < 2 )  
    System.out.println( m );
```

Which of the following are printed to standard output?

- (a) 0
- (b) 1
- (c) 2
- (d) 3
- (e) Nothing and an exception is thrown

5. Consider the code fragment below:

```
outer: for( int i = 1; i <3; i++ )  
    inner: for( j = 1; j < 3; j++ )  
    {  
        if( j==2 )  
            continue outer;  
        System.out.println( “i = ” + i +”, j = ” + j );  
    }
```

Which of the following would be printed to standard output?

- (a) i = 1, j = 1
- (b) i = 1, j = 2
- (c) i = 1, j = 3
- (d) i = 2, j = 1
- (e) i = 2, j = 2
- (f) i = 2, j = 3
- (g) i = 3, j = 1
- (h) i = 3, j = 2

Answers:

1(c,d,e,f,g)

2(a,b)

3(b)

4(b,c)

5(a,d)

Theory Questions:

1. Does Java has “goto”?
2. What are labeled loops.

Programming Exercises:

1. Write a program to check whether a given year is leap or not.
2. Write a program to print the number of days in a given month.(using switch)
3. Write a program to print Fibonacci series.
4. Write a program to check whether a given number is prime or not.
5. Write a menu driven program to calculate addition, subtraction, multiplication and division of two numbers.
 - 1 for Add
 - 2 for Subtract
 - 3 for Multiplication
 - 4 for Divide
 - 5 for Exit (hint: do while and switch)
6. Write a program to print the reverse of a number. (hint: 1234 will result in 4321).
7. Write a program to find the maximum of the given three numbers.
8. Write a program to print first 10 even numbers using jump statement.



ISBN 13: 978-1500730413.



CHAPTER

∞ 4 ∞

(Scanner Class & Arrays)
(Command Line Arguments)

Introduction-

Scanner class:-

This is a new class in Java added in JDK1.5-6-7 version to take input of primitive data type from the user. It is very easy to take input using this class as we have to only create an object of this class and have to call a particular method. This class is available in java.util package.

```
public java.util.Scanner(java.io.InputStream);  
public java.util.Scanner(java.io.File)    throws java.io.FileNotFoundException;  
public java.util.Scanner(java.lang.String);  
public void close();  
public boolean hasNext();  
public java.lang.String next();  
To input a string not containing any space.  
public java.lang.String nextLine();  
To input a string which may contain space also.  
public boolean hasNextBoolean();  
public boolean nextBoolean();  
public boolean hasNextByte();  
public byte nextByte();  
public boolean hasNextShort();  
public short nextShort();
```

```
public boolean hasNextInt();
```

To check whether next value in input buffer is int or not.

```
public int nextInt();
```

To input a integer value.

```
public boolean hasNextLong();
```

```
public long nextLong();
```

```
public boolean hasNextFloat();
```

```
public float nextFloat();
```

```
public boolean hasNextDouble();
```

```
public double nextDouble();
```

Example 4.1:

```
1.      import java.util.*;
2.      class SumInput
3.      {
4.          public static void main(String args[])
5.          {
6.              Scanner sc=new Scanner(System.in);
7.              int a,b,sum;
8.              System.out.print("Enter Ist number");
9.              a = sc.nextInt();
10.             System.out.print("Enter 2nd number");
11.             b = sc.nextInt();
12.             sum = a+b;
```

```
13.           System.out.println("Sum is " + sum);
14.         }
15.     }
```

Output:

Enter Ist number 45

Enter 2nd number 55

Sum is 100

Array-

An array in Java is an ordered collection of the similar type of variables or data items. Java allows arrays of any dimension. An array in Java is a bit different from C/C++. We can not specify the size of the array at the time of declaration. The memory allocation is always dynamic. Every array in Java is treated as an object with one special attribute length, which specifies the number of elements in the array.

1. One-Dimensional array:-

Declaration-

```
type array_name[ ];          // No memory allocation takes place in declaration.
```

Or

```
type [ ] array_name;
```

It is important to note that the declaration does not actually create an array. It only declares a reference that can denote an array object.

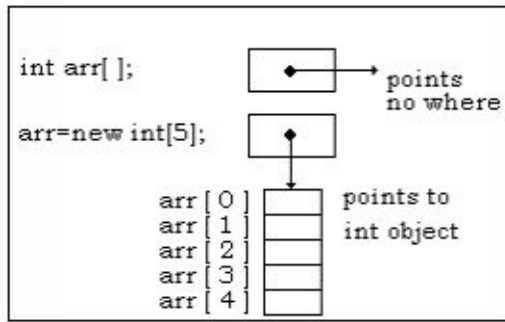
```
int a1 [ ], a2 ;
```

```
int [ ] a3, a4 ;
```

These two declarations declare a1, a3 and a4 to be reference variables that can denote array of int, but the variable a2 can not denote an array of int value. It is simply an int variable. When the [] notation follows the type, all variable in the declaration are array. Otherwise the [] notation must follows each individual array name in the declaration.

Memory allocation with the help of new operator:-

```
array_name = new type[SIZE];          // SIZE can be a constant or a variable.
```



(Creation of an array in memory)

Note1- All the elements of array will be automatically initialized to zero but to initialize the array with different values we can initialize the array in declaration itself. Array declaration, memory allocation and initialization steps can be combined into one:-

```
int a[]={5,7,1};
```

Note2-Array index starts from zero.

Note3:-In Java, all arrays store the allocated size in a variable named length. We can access the length of the array using the attribute length:

“array_name.length”.

```
Note4-    int a[],b[];
           a = b = new int[5];
```

In the above steps only one array is created by new, the reference of this array will be stored first in b & then a. So both a & b are pointing to a single array. Assigning a reference does not create a copy of the object.

Example 4.2 Sum of all the numbers stored in an Array.

```
1.    class SumArr
2.    {
3.        public static void main(String args[])
4.        {
```

```

5.         int a[] = {10,50,20,40,30};
6.         int n = a.length, sum = 0;
7.         for(int i = 0; i < n; i++)
8.             sum += a[i];
9.         System.out.println("Sum of numbers is " + sum);
10.        }
11.    }

```

Output: Sum of numbers is 150

Example 4.3 Program to sort an array using selection sorting technique.

```

1.     import java.util.Scanner;
2.     class Sort
3.     {
4.         public static void main(String args[])
5.         {
6.             int n;
7.             Scanner sc = new Scanner(System.in);
8.             System.out.println("Enter how many elements");
9.             n = sc.nextInt();
10.            int a[] = new int[n];
11.            //input array
12.            for(int i = 0; i < n; i++)
13.            {
14.                System.out.print("Enter element " + (i+1) + " ");
15.                a[i] = sc.nextInt();
16.            }
17.            //sorting
18.            for(int i = 0; i < n-1; i++)
19.                for(int j = i+1; j < n; j++)
20.                    if(a[i] > a[j])
21.                        {

```



```

22.                int t=a[i];
23.                a[i] = a[j];
24.                a[j] = t;
25.                }
26.                //output
27.                for(int i=0;i<n;i++)
28.                    System.out.println(a[i]);
29.            }
30.        }

```

2. Two-Dimensional Array-

Declaration:-

```

type array_name[][];
or
type[][] array_name;

```

Memory allocation:-

```

array_name = new type[ROWS][COLUMNS];
//ROWS & COLUMNS can be constants or variables

```

So far we have discussed the array variables that can store a list of values. There will be situation where a table of value will have to be stored. Consider the following data table, which show the value of sales of three items by sales person.

The table contains total 12 value, three in each line. We can think of this table as a matrix consisting of four row and three columns. Each row represents the value of sales by a particular sales person and each columns represent the value of sales of a particular item.

Example 4.4 Program to print the transpose of a matrix

```

1.     import java.util.*;
2.     class Transpose
3.     {
4.         public static void main(String args[])
5.         {
6.             Scanner sc=new Scanner(System.in);
7.             int r,c;
8.             System.out.print("Enter how many rows");
9.             r=sc.nextInt();
10.            System.out.print("Enter how many cols.");
11.            c=sc.nextInt();
12.
13.            int m[][]=new int[r][c];
14.            //input
15.            for(int i=0;i<r;i++)
16.                for(int j=0;j<c;j++)
17.                    {
18.                        System.out.print("Enter" + "element " + (i+1) + "," + (j+1));
19.                        m[i][j]=sc.nextInt();
20.                    }
21.            //transpose
22.            for(int i=0;i<c;i++)
23.                {
24.                    for(int j=0;j<r;j++)
25.                        System.out.print(m[j][i]+"");
26.                    System.out.println();
27.                }
28.        }
29.    }

```

Example 4.5 Program to multiply two matrices into a 3rd matrix.

```
1.     import java.util.*;
2.     class MatrixMult
3.     {
4.         public static void main(String args[])
5.         {
6.             Scanner sc=new Scanner(System.in);
7.             int r1,c1;
8.             System.out.print("Enter how many rows");
9.             r1=sc.nextInt();
10.            System.out.print("Enter how many cols.");
11.            c1=sc.nextInt();
12.            int r2,c2;
13.            System.out.print("Enter how many rows");
14.            r2=sc.nextInt();
15.            System.out.print("Enter how many cols.");
16.            c2=sc.nextInt();
17.            if(c1 != r2)
18.            {
19.                System.out.println("Can't mult");
20.                System.exit(1);
21.            }
22.            int m1[][]=new int[r1][c1];
23.            //input m1
24.            for(int i=0;i<r1;i++)
25.                for(int j=0;j<c1;j++)
26.                {
27.System.out.print("Enter" + "element " + (i+1) + "," + (j+1));
28.                    m1[i][j]=sc.nextInt();
29.                }
30.            int m2[][]=new int[r2][c2];
31.            //input m2
32.            for(int i=0;i<r2;i++)
```

```

33.             for(int j=0;j<c2;j++)
34.             {
35. System.out.print("Enter" + "element " + (i+1) + "," + (j+1));
36.             m2[i][j]=sc.nextInt();
37.             }
38. int m3[][]=new int[r1][c2];
39. //matrix mult
40. for(int i=0;i<r1;i++)
41.     for(int j=0;j<c2;j++)
42.         for(int k=0;k<c1;k++)
43.             m3[i][j]+=m1[i][k]*m2[k][j];
44. //Output m3
45. for(int i=0;i<r1;i++)
46.     {
47.         for(int j=0;j<c2;j++)
48.             System.out.print(m3[i][j]+"\\t");
49.         System.out.println();
50.     }
51.     }
52. }

```

3.Variable size Array:

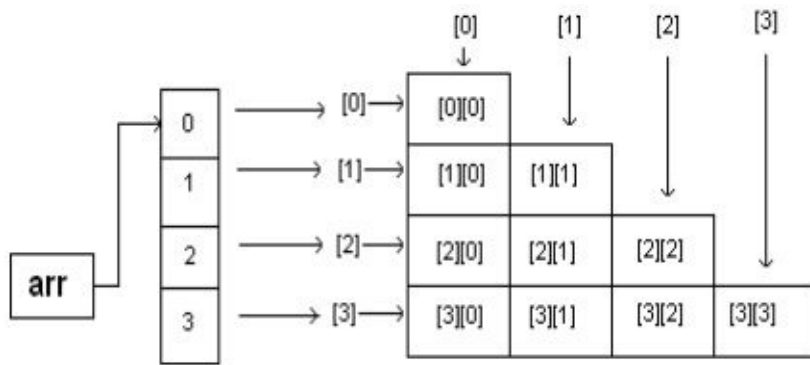
Array treat multidimensional array as “array of array” it is possible to declare a two-d array as follows

```

int arr[ ] [ ] = new int [4][ ];
arr[0] = new int [1];
arr[1] = new int [2];
arr[2] = new int [3];
arr[3] = new int [4];

```

This means total 4 rows of array and its columns starts from 1 and increase one by one. so they are represented in the memory as the follows:



(Memory allocation)

Note1- All the elements of array will be automatically initialized to zero but to initialize the array with different values we can initialize the array in declaration itself. Array declaration, memory allocation and initialization steps can be combined into one:-

```
int a[][] = { {5,7,1}, {1,7,9}, {4,5,7}, {1,1,1} };
```

Note2- In the above array `a.length` will show the number of rows i.e. 4 and `a[0].length` will show number of columns in 0th row i.e. 3

Command Line Arguments-

Sometimes we will want to pass information into a program when we run it. This is accomplished by passing command-line arguments to `main()`. A command line argument is the information that directly follows the program's name on the command line when it is executed.

To access the command-line arguments inside a Java program we have to use String array passed to `main()`.

The JVM calls the `main()` method and passes the command line argument to it as an array of string. The length of the array (i.e the number of the command line arguments) is obtained using attribute `length` in the above example.

The for loop displays the command line argument on the console/monitor.

Example 4.6 Program to input names from command line arguments.

```
1.      class Hello
2.      {
3.          public static void main(String args[])
4.          {
5.              for(int i=0;i<args.length;i++)
6.                  System.out.println("Hello " + args[i]);
7.          }
8.      }
```

javac Hello.java

java Hello Ravi Ajay Vijay

Output: Hello Ravi

 Hello Ajay

 Hello Vijay

Example 4.7 Program two print sum of numbers input through command line arguments.

```
1.      class SumCmd
2.      {
3.          public static void main(String args[])
4.          {
5.              int sum=0;
6.              for(int i=0;i<args.length;i++)
7.                  sum+=Integer.parseInt(args[i]);
8.              System.out.println("Sum is " + sum);
9.          }
10.     }
```

javac SumCmd.java

java SumCmd 5 6

Output:

 Sum is 11

Example 4.8 Program to sort an array input from command line.

```
1.     class SortCmd
2.     {
3.         public static void main(String args[])
4.         {
5.             int n=args.length;
6.             int a[ ]=new int[n];
7.             //copy args array to a array
8.             for(int i=0;i<n;i++)
9.                 a[i] = Integer.parseInt(args[i]);
10.            //sorting
11.            for(int i=0;i<n-1;i++)
12.                for(int j=i+1;j<n;j++)
13.                    if(a[i]>a[j])
14.                        {
15.                            int t=a[i];
16.                            a[i]=a[j];
17.                            a[j]=t;
18.                        }
19.            //Output Array
20.            for(int i=0;i<n;i++)
21.                System.out.print(a[i]+" ");
22.        }
23.    }
```

javac SortCmd.java

java SortCmd 5 4 1 3 2

Output: 1 2 3 4 5

Multiple choice questions:

1. If MyProg.java were compiled as an application and then run from the command line as: **java MyProg I like tests**
what would be the value of args[1] inside the main() method?
- (a) MyProg
 - (b) "I"
 - (c) "like"
 - (d) 3
 - (e) 4
 - (f) null until a value is assigned
2. After the declaration: **char[] c = new char[100];**
what is the value of c[50]?
- (a) 50
 - (b) 49
 - (c) '\u0000'
 - (d) '\u0020'
3. Which of the following are legal declarations of a two-dimensional array of integers?
- (a) int[5][5]a = new int[][];
 - (b) int a = new int[5,5];
 - (c) int[][]a = new int[5][5];
 - (d) int[][]a = new [5]int[5];
4. Which of the following are correct methods for initializing the array "dayhigh" with 7 values?
- (a) int dayhigh = { 24, 23, 24, 25, 25, 23, 21 };
 - (b) int dayhigh[] = { 24, 23, 24, 25, 25, 23, 21 };
 - (c) int[] dayhigh = { 24, 23, 24, 25, 25, 23, 21 };
 - (d) int dayhigh [] = new int[24, 23, 24, 25, 25, 23, 21];
 - (e) int dayhigh = new[24, 23, 24, 25, 25, 23, 21];
5. Which of the following is a correct statement?
- (a) A Java array is an Object.
 - (b) Memory allocation for arrays in Java is always dynamic.
 - (c) Java array is initialized by default values of the type of its elements.
 - (d) All of the above.

Answer's: 1(c) 2(e) 3(c) 4(b, c) 5(d)

Programming Exercise:

1. Write a program to search an element in an array using linear search.
2. Write a program to search an element in an array using binary search.



CHAPTER

∞ 5 ∞

(Class and Objects)

Introduction-

Defining a class-

Class represents a ADT (Abstract Data Type). It acts like a template using which we can create multiple objects (instances). A class declaration only creates a template, it does not create an actual object. A class creates a new data type that can be used to create objects. That is, a class creates a logical framework that defines the relationship between its members. When we declare an object of a class, we are creating an instance of that class. Thus a class is a logical construct. An object has physical reality. (That is, an object occupies space in memory.)

```
modifiers class <classname>
```

```
{
```

```
    modifiers type variables;
```

```
    .
```

```
    .
```

```
    .
```

```
    modifiers type methodName1(parameter-list)
```

```
    {
```

```
        <body of the method>
```

```
    }
```

```
    .
```

```
    .
```

```
}
```

Declaring Objects-

```
Student s1;
```

In the above example s1 is not an object. In C++ s1 will be treated as object but in Java s1 is only a variable which can hold reference to an object of type Student. This variable will hold garbage or null reference until we assign reference of some object of class Student.

It does not yet point to an actual object. Any attempt to use this variable at this point will result in a compile-time error. It is just like a pointer to an object in C/C++. If this variable is declared inside any block or a method then it is a local variable and will not be initialized to null, but if it is declared in a class then it will be an instance variable and will automatically be initialized to null.

Allocating Memory-

In Java memory is allocated dynamically with the help of new operator.

```
s1 = new Student();
```

In the above example new operator will allocate memory for an object of class Student and return its reference, which is then assigned to reference type variable s1. It will also call a default constructor to initialize member variables.

The above two statements can be combined:-

```
Student s1 = new Student();
```

Note1- new allocates memory for an object during runtime. If new is unable to allocate memory (because memory is finite.) then it generate a run time exception/error.

Note2- If we assign a reference variable to another reference variable then only reference (address) will be transferred. There will not be any duplicate copy of the object. For example if we create another reference variable s2 (Student s2) and then (s1 = s2). This will not create duplicate object, same object is referenced by s1 and s2.

Any changes made in s2 will also be reflected in s1. No memory allocation is done with this assign. After a few steps if we assign null in s1 then that object will not be destroyed as its reference is with object s2. But if both s1 and s2 are assigned with a null value then that object will be destroyed by the garbage collector and the memory occupied by that object will be freed.

Example 5.1 Class containing member variables only.

```
1.      class A
2.      {
3.          int x;
4.          int y;
5.      }
6.      class ClassTest1
7.      {
8.          public static void main(String args[])
9.          {
10.             A a1,a2; //Reference Variables
11.             a1 = new A(); //Object Createion
12.             a2 = new A();
13.             a1.x = 10;
14.             a1.y = 20;
15.             a2.x = 5;
16.             a2.y = 6;
17.             System.out.println(a1.x + "\t" + a1.y);
18.             System.out.println(a2.x + "\t" + a2.y);
19.         }
20.     }
```

Output:

```
10      20
      5      6
```

Example 5.2 class containing member variables & member methods.

```
1.     class A
2.     {
3.         private int x;
4.         private int y;
5.         void setdata(int x1, int y1)
6.         {
7.             x = x1;
8.             y = y1;
9.         }
10.        void display()
11.        {
12.            System.out.println(x + "\t" + y);
13.        }
14.    }
15.    class ClassTest2
16.    {
17.        public static void main(String args[])
18.        {
19.            A a1 = new A();
20.            A a2 = new A();
21.            //a1.x=10; error as x is private
22.            a1.setdata(10,20);
23.            a2.setdata(5,7);
24.            a1.display();
25.            a2.display();
26.        }
27.    }
```

Output:

10 20

Note- We can declare the object of a class outside the class as well as inside the class.

Example 5.3 Defining objects of a class in itself.

```
1.      class A
2.      {
3.          private int x;
4.          private int y;
5.          void setdata(int x1, int y1)
6.          {
7.              x = x1;
8.              y = y1;
9.          }
10.         void display()
11.         {
12.             System.out.println(x + "\t" + y);
13.         }
14.         public static void main(String args[])
15.         {
16.             A a1 = new A();
17.             A a2 = new A();
18.             a1.setdata(10,20);
19.             a2.setdata(5,7);
20.             a1.display();
21.             a2.display();
22.         }
23.     }
```

Output:

10 20

Modifiers / Visibility Labels for members of a class-

The visibility modifiers are applicable only on members of a class not on local variables.

1. **private-** If we specify the modifier private with a member variable or method then that member will not be visible outside the class in which it is declared. This will hide the member of a class from other classes.
2. **public-** If we specify the modifier public with a member variable or method then that member will be visible to all the classes. This member can be accessed even outside the package. main() method is always defined as public because the main() method is accessed by the JVM which is outside the class in which main() method is defined.
3. **protected-** A member declared as protected can be accessed from all the classes belonging to the same package. Protected members can also be accessed from any sub class of other packages.
4. **default / no modifier-** If no visibility modifier is specified before a member declaration then that member can be accessed from all the classes in the same package. That member can not be accessed outside the package.

Visibility Modifiers for a class or interface-

1. **public-** If a class is to be visible to all the classes irrespective of their package, then it must be declared as public by specifying the modifier public, which should appear before the keyword class.
2. **default / no modifier:-** In the absence of any access/visibility modifier before the class, its visibility is only within the package(group of classes) in which it is defined.

this Keyword:-

this is a reference variable which stores the reference of the object currently used to call the method. The reference of this is replaced by the reference of the calling object at run-time. Sometimes a method will need to refer to the object that invoked it. This can be done with the help of this. this can be used inside any method to refer to the current object.

Note: local variable hides the instance variable, so we have to use this keyword explicitly.

Instance Variable Hiding-

As we know, it is illegal in Java to declare two local variables with the same name inside the same or enclosing scopes. But we can have local variables (parameters of member methods) having same name as of instance member variables of class.

When a local variable has the same name as an instance variable, the local variables hides the instance variable. But this keyword allows us to refer to instance variables even if local variable hides it.

Example 5.4 use of *this* keyword

```
1.      class A
2.      {
3.          private int x;
4.          private int y;
5.          void setdata(int x, int y)
6.          {
7.              this.x = x; //this.x is the class member & x is local variable
8.              this.y = y;
9.          }
10.         void display()
11.         {
12.             System.out.println(x + "\t" + y);
13.         }
14.     }
15.     class ClassTest4
16.     {
```



```

17.         public static void main(String args[])
18.         {
19.             A a1 = new A();
20.             A a2 = new A();
21.             a1.setdata(10,20);
22.             a2.setdata(5,7);
23.             a1.display();
24.             a2.display();
25.         }
26.     }

```

Output:

```

10      20
      5      7

```

Method Overloading:-

We can have more than one method with the same name as long as they differ either in numbers of parameters or type of parameters or order of parameters. This is called method overloading.

While calling an overloaded method it is possible that type of the actual parameters passed may not match exactly with the formal parameters of any of the overloaded methods. In that case parameter are promoted to next higher type till a match is found. If no match is found even after promoting the parameters then a compilation error occurs.

Example 5.5 In this example there are two setdata() methods having same name but having different arguments, so this is method overloading.

```

1.     class A
2.     {
3.         private int x,y;
4.         void setdata(int x1)

```

```

5.      {
6.          x = y = x1;
7.      }
8.      void setdata(int x1, int y1)
9.      {
10.         x = x1;
11.         y = y1;
12.     }
13.     void display()
14.     {
15.         System.out.println(x + "\t" + y);
16.     }
17. }
18. class ClassTest5
19. {
20.     public static void main(String args[])
21.     {
22.         A a1 = new A();
23.         a1.setdata(5);
24.         a1.display();
25.         A a2 = new A();
26.         a2.setdata(10,20);
27.         a2.display();
28.     }
29. }

```

Output:

```

5      5
      10      20

```

Constructors-

It is very common requirement to initialize an object immediately after creation. We can define instance methods for this purpose but they have to be invoked explicitly.

Java has a solution for this requirement. Java allows objects to initialize themselves when they are created using constructors. It has the same

The syntax of the constructors is very similar to that of instance methods. They have the same name as the class and do not have any return type.

This is because the implicit return type of class's constructor is the class itself. Constructors can be overloaded just like methods.

When operator new is used to create an instance/object of a class, JVM allocates memory for the object, then initializes the instance variables to their default initial values, and then calls the appropriate constructor to initialize the instance variables.

Note-

The name constructor is a bit confusing. It appears as if the purpose of the constructor is to create an object/instance. The object is created and instance variables and static variables are initialized to their default initial values before constructor is called.

So the purpose of the constructor is to initialize the instance variables with values other than the default values.

Type of constructors-

Default Constructor:- Every class has a default constructor (if no explicit constructor is defined) that does not take any argument and its body does not have any statements. The compiler generates the default constructor automatically.

The compiler stops generating default constructor as soon as we add our own constructor. When we do not create any constructor in the class then JVM will create the default constructor and initialize the instance variable with default values null or zero.

Example 5.6 use of default constructor

```
1.     class A
2.     {
3.         private int x;
4.         private int y;
5.         void display()
6.         {
7.             System.out.println(x + "\t" + y);
8.         }
9.     }
10.    class ClassTest6
11.    {
12.        public static void main(String args[])
13.        {
14.            A a1 = new A();
15.            a1.display();
16.        }
17.    }
```

Output:

```
0      0
```

Note:-Here we have not created any constructor then JVM create automatically default constructor after creating object and initialize the instance variables from 0 or null.

Default Zero argument constructors- We can replace the default constructor with our own zero argument constructor. This will allow us to initialize the instance variables to any value.

Parameterized Constructors:- A constructor which takes parameters is called as parameterized constructors.

Note- It is possible to overload the constructor just like methods. It is called as constructor overloading.

Example 5.7 use of constructor.

```
1.     class A
2.     {
3.         private int x;
4.         private int y;
5.         A() //Zero argument constructor
6.         {
7.             x = y = 0;
8.         }
9.         A(int x1) //Parameterized one argument constructor
10.        {
11.            x = y = x1;
12.        }
13.        A(int x1, int y1) //Parameterized two argument constructor
14.        {
15.            x = x1;
16.            y = y1;
17.        }
18.        void display()
19.        {
20.            System.out.println(x + "\t" + y);
21.        }
22.    }
23.    class ClassTest7
24.    {
25.        public static void main(String args[])
26.        {
27.            A a1 = new A();
28.            a1.display();
29.            A a2 = new A(5);
30.            a2.display();
```

```

31.         A a3 = new A(4,7);
32.         a3.display();
33.     }
34. }

```

Output:

```

0      0
5      5
4      7

```

Example5.8:

```

1.     import java.util.Scanner;
2.     class Complex
3.     {
4.         private int real,imag;
5.         Complex()//zero arg. constructor
6.         {
7.             real=imag=0;
8.         }
9.         Complex(int real, int imag)
10.        {
11.            this.real=real;
12.            this.imag=imag;
13.        }
14.        void getdata()
15.        {
16.            Scanner sc=new Scanner(System.in);
17.            System.out.print("Enter real");
18.            real=sc.nextInt();
19.            System.out.print("Enter imag");
20.            imag=sc.nextInt();
21.        }

```

```

22.         void display()
23.         {
24.             if(imag>=0)
25.                 System.out.println(real+" "+imag+"i");
26.             else
27.                 System.out.println(real+" "+imag+"i");
28.         }
29.         Complex sum(Complex c)
30.         {
31.             Complex t=new Complex();
32.             t.real=real+c.real;
33.             t.imag=imag+c.imag;
34.             return t;
35.             //         or
36.             //         return new Complex(real+c.real,imag+c.imag);
37.         }
38.         Complex mult(Complex c)
39.         {
40.             Complex t=new Complex();
41.             t.real=real*c.real-imag*c.imag;
42.             t.imag=real*c.imag+imag*c.real;
43.             return t;
44.         }
45.         public static void main(String args[])
46.         {
47.             Complex c1=new Complex();
48.             Complex c2=new Complex();
49.             Complex c3=null,c4=null;
50.
51.             c1.getdata();
52.             c2.getdata();
53.             c3=c1.sum(c2);

```

```
54.         System.out.print("Sum is ");
55.         c3.display();
56.         c4=c1.mult(c2);
57.         System.out.print("Product is ");
58.         c4.display();
59.     } }
```

finalize () Method-

Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handler, then we might want to make sure these resources are freed before an object is destroyed.

To handle such situations, Java provides a mechanism called finalization. By using finalization we can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector. `finalize ()` is only called just prior to garbage collection. It is not called when an object goes out of scope. In C++ the concept of destructor function is used for finalization but it is not exactly same as `finalize ()` in Java.

```
protected void finalize()
{
    // finalization code here.
}
```

Note-Instead of protected modifier we can also use public.

Type of Variables-

Java has three kinds of variables:

1. Instance variables-

Variable declared in a class as a member outside all member methods are known as instance variable. There will be as many copies of that variable as there are number of objects.

There is no need to initialize instance variable in Java because these variables are initialized as soon as the object is created and the memory is allocated with the new operator.

The variables are initialized according to their types. All the numeric variables are initialized automatically with 0, Boolean variable with false, and reference variables (objects) with null value.

We can access these variables by using object name and the dot (.) operator. (Object reference variable.instance variable name). We can not use operator to separate object reference variable and the instance variable as in C/C++.

2. Local Variables:-

Variables declared inside a method is known as Local variable. It is not same as instance variable. There is no automatic initialization for the Local variable.

3. Static Variables:-

Static variables are also declared outside methods/blocks like instance variables. But the static variables make use of the modifier static before the data type. Static variables are global to a class and all of its instances (objects).

They are useful for keeping track of global states. For example, a static variable count in a class can store the number of instances/objects of the class created in the program at any instance of time.

The objects can communicate using static variables just like C functions communicate through global variables. For static variables there is only one copy of the variable irrespective of number of instances/objects created in the program, which is shared across all the instances.

The dot (.) operator is used to access the class variables also, but we can access the static variable using class name as well as object name. If we declare to different objects the both will point to same copy the static variable.

Static variable is initialized automatically with their default values (zero, false or null) as soon as the class is loaded / used.

They can only call other static methods. They must only access static data. They can not refer to this or super in any way. (The key word super relates to inheritance and is described later.)

Example 5.9

```
1.      class A
2.      {
3.          int x;      //instance variable
4.          static int y; //static or class variable
5.      }
6.      class ClassTest8
7.      {
8.          public static void main(String args[])
9.          {
10.             A a1 = new A();
11.             A a2 = new A();
12.             A a3 = new A();
13.             a1.x = 10;
14.             a1.y = 20;
15.             a2.x = 11;
16.             a2.y = 21;
17.             a3.x = 12;
18.             a3.y = 22;
19.             System.out.println(a1.x + "\t" + a1.y);
20.             System.out.println(a2.x + "\t" + a2.y);
21.             System.out.println(a2.x + "\t" + a3.y);
22.         }
23.     }
```

Output:

```
10      22
      11      22
      12      22
```

Type of Methods:-

1. Instance methods.
2. Static methods.

Instance methods:-

These are same as C++ functions. Instance methods can be invoked only through object. If we call a instance method inside a static method of same class then also we have to use object name.

Static Methods:-

Static methods can be invoked using object as well as class name. Static methods can access only static members. Static methods can be called directly (without object or class name) from a static method of same class.

Example 5.10

1. class A
2. {
3. int x;
4. static int y;
5. static int sum(int a, int b)
6. {
7. //y = 5; no error static method can access static variable
8. //x = 10; error static method can't access instance variable
9. //A a1 = new A();
10. //a1.x = 10; No error as static method can access instance variable
11. //through objects.
12. //this.x = 10; error as static method can't access this or super keyword

```

13.         int c;
14.         c = a+b;
15.         return(c);
16.     }
17.     static float avg(int a, int b)
18.     {
19.         return (float)(a+b)/2;
20.     }
21. }
22. class ClassTest9
23. {
24.     public static void main(String args[ ])
25.     {
26.         System.out.println(A.sum(5,6)); // static methods can be called through
27.                                         // class name
28.         System.out.println(A.avg(5,6));
29.         A a1 = new A();
30.         System.out.println(a1.sum(5,6)); // static methods can also be called
31.                                         // through objects
32.     }
33. }

```

Output:

```

11
5.5
11

```

Initialize & static Block- Initialize block is used to initialize instance member variables of the class & static block is used to initialize static member variables of the class. But constructor is called after these blocks.

Example 5.11:

```

1.     class A
2.     {
3.         int x;
4.         static int y;

```

```

5.
6.         { //Inititalize block to initialize instance variables
7.             x = 10;
8.         }
9.
10.        static //static block to initialize static variables
11.        {
12.            y = 5;
13.        }
14.    }
15.    class ClassTest10
16.    {
17.        public static void main(String args[])
18.        {
19.            A a1 = new A();
20.            System.out.println(a1.x + "\t" + a1.y);
21.        }
22.    }

```

Output:

```

10      5

```

Final Variables-

If a variable is declared as final then we can not change its value final (**double PI=3.141;**) final variable do not occupy memory on a per instance basis. A final variable is a constant variable its value can not be change. A final variable reference type can not change its reference.

Example 5.12

```

1.        class A
2.        {
3.            void m1()
4.            {
5.                System.out.println("Inside m1()");
6.            }
7.        }
8.        class ClassTest11

```

```

9.      {
10.         public static void main(String args[])
11.         {
12.             final int x=10; //can not change
13.             final int y; // Variable y not initialized so we can initialize y later
14.             y=20; // Now y is initialize so after this we can't change its value
15.             // y=30; Error
16.             System.out.println(x);
17.             System.out.println(y);
18.             final A a1=new A(); // a1 is a constant reference so we can not
19.             // assign reference of any other object in f1.
20.             a1.m1();
21.             A a2 = new A();
22.             // a1 = a2; error
23.         }
24.     }

```

Output: 10
 20
 Inside m1()

Argument Passing Mechanism-

1. **Call by Value.**
2. **Call by reference.**

When a primitive type (int, float, long etc.) is passed to a method, it is done by use of call-by-value approach. In case of objects what is actually passed is an object reference.

An object reference is also passed by using call-by-value approach. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

For example if we pass a reference variable a1 which stores a reference of an object into a function and in function a2 is the formal parameter then a1 and a2 both points to the same object.

Any changes in the value of the object through a2 will also be reflected in a1. But if we assign a new reference in a2 then that reference will not be automatically copied in a1.

Example 5.13

```
1.      class Box
2.      {
3.          private int feet;
4.          private int inches;
5.          Box()
6.          {
7.              feet = inches = 0;
8.          }
9.          Box(int feet, int inches)
10.         {
11.             this.feet = feet;
12.             this.inches = inches;
13.         }
14.         void display()
15.         {
16.             System.out.println("Feet is " + feet + "\n" + "Inches is " + inches);
17.         }
18.         void swap(Box obj)
19.         {
20.             int t;
21.             t = feet;
22.             feet = obj.feet;
23.             obj.feet = t;
24.
```

```

25.         t = inches;
26.         inches = obj.inches;
27.         obj.inches = t;
28.     }
29. }
30. class ClassTest12
31. {
32.     public static void main(String args[])
33.     {
34.         Box b1 = new Box(5,6);
35.         Box b2 = new Box(7,4);
36.         b1.swap(b2); //Object is passed by reference.
37.         b1.display();
38.         b2.display();
39.     }
40. }

```

Output:

Feet is 7

Inches is 4

Feet is 5

Inches is 6

Nested and Inner Classes-

It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class.

A nested class has access to the members, including private members of the class in which it is nested. However the enclosing class does not have access to the members of the nested class.

There are two types of nested classes static and non static. A static nested class is one which has the static modifier applied. Because it is static, it must access the members of its enclosing class through an object. That is, it can not refer to members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

A non-static nested class has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

Example 5.14

```
1.      class Outer
2.      {
3.          class Inner
4.          {
5.              int member_inner=7;
6.              Inner()
7.              {
8.                  member_outer = 5;
9.                  System.out.println(member_outer);
10.             }
11.         }
12.         private int member_outer;
13.         Outer()
14.         {
15.             //  member_inner = 9;    error
16.             Inner obj = new Inner();
17.             System.out.println(obj.member_inner);
18.         }
19.     }
20.     class ClassTest13
21.     {
22.         public static void main(String args[])
```

```

23.         {
24.             Outer out_obj = new Outer();
25.         }
26.     }

```

Output:

```

5
7

```

Example 5.15:

```

1.     class Outer
2.     {
3.         static class Inner
4.         {
5.             int member_inner=7;
6.             Inner()
7.             {
8.                 // member_outer = 5; error
9.                 // System.out.println(member_outer); error
10.                Outer obj = new Outer();
11.                obj.member_outer = 5;
12.                System.out.println(obj.member_outer);
14.            }
15.        }
16.        private int member_outer;
17.        void prn()
18.        {
19.            Inner obj = new Inner();
20.            System.out.println(obj.member_inner);
21.        }
22.    }

```

```
23.     class ClassTest14
24.     {
25.         public static void main(String args[])
26.         {
27.             Outer out_obj = new Outer();
28.             out_obj.prn();
29.         }
30.     }
```

Output:

5

7

Class Random

Example 5.16:

```
1.     import java.util.Random;
2.     class RandomTest
3.     {
4.         public static void main(String args[])
5.         {
6.             Random r1 = new Random();
7.             for(int i=1;i<=10;i++)
8.                 System.out.println(Math.abs(r1.nextInt()));
9.         }
10.    }
```

Theory Questions:

1. If we want a member variable to not be accessible outside the current class at all, what keyword should precede the name of the variable when declaring it?
2. In Java, how are objects / values passed around.
3. Consider the code below:

```
public static void main( String args[] )
{
    int a = 5;
    System.out.println( cube( a ) );
}
int cube( int theNum )
{
    return theNum * theNum * theNum;
}
```

Explain the importance of “static” keyword
What will happen when we attempt to compile and run this code?

- a) It will not compile because cube is already defined in the java.lang.Math class.
- b) It will not compile because cube is not static.
- c) It will compile, but throw an arithmetic exception.
- d) It will run perfectly and print “125” to standard output.

4. What does a static inner class mean? How is it different from any other static member
5. How do we declare constant values in java
6. What is the meaning of “final” keyword?
7. What is the class variables?

Answer:

When we create a number of objects of the same class, then each object will share a common copy of variables. That means that there is only one copy per class, no matter how many objects are created from it. Class variables or static variables are declared with the static keyword in a class.

These variables are stored in static memory. Class variables are mostly used for constants, variable that never change its initial value. Static variables are always called by the class name.

This variable is created when the program starts i.e. it is created before the instance is created of class by using new operator and gets destroyed when the programs stops. The scope of the class variable is same as instance variable.

The class variable can be defined anywhere at class level with the keyword static. Its initial value is same as instance variable. When the class variable is defined as int then its initial value is by default zero, when declared boolean its default value is false and null for object references. Class variables are associated with the class, rather than with any object.

8. Explain garbage collection.

Answer: Garbage collection is one of the most important feature of Java. Garbage collection is also called automatic memory management as JVM automatically removes the unused variables/objects (value is null) from the memory. User program can't directly free the object from memory, instead it is the job of the garbage collector to automatically free the objects that are no longer referenced by a program.

Every class inherits **finalize()** method from **java.lang.Object**, the **finalize()** method is called by garbage collector when it determines no more references to the object exists.

In Java, it is good idea to explicitly assign **null** into a variable when no more in use. In Java on calling **System.gc()** and **Runtime.gc()**, JVM tries to recycle the unused objects, but there is no guarantee when all the objects will garbage collected.

9. Can we call one constructor from another if a class has multiple constructors?

Yes. Use **this()** to call a constructor from an other constructor.

10. What's the difference between constructors and normal methods? Constructors must have the same name as the class and can not return a value. They are only called once while regular methods could be called many times and it can return a value or can be void.

11. How we can force the garbage collection? Garbage collection automatic process and can't be forced. We could request it by calling **System.gc()**. JVM does not guarantee that GC will be started immediately.

12. Which of the following is not a correct statement.

- (a) Local variable in Java are always initialized by default values.
- (b) Local variables in Java must be initialized before use.
- (c) Local variables in an inner block can not have the same name as a local variable in outer block.
- (d) Local variables in two blocks as the same level can have same name.





CHAPTER

∞ 6 ∞

(Inheritance)

Introduction-

When the properties of the one class is transferred into another class then it's said to be an inheritance. In this way the inherited class is called to be a super class or base class or parent class and inheriting class is called to be a child class sub class derived class.

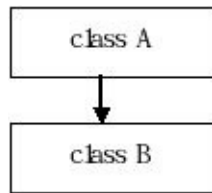
Inheritance allows us to extend an existing class (Base class) by adding the additional features. It encourages the code reusability, which help in reducing the code size although it may lead to complexity in some cases.

As an example if we have completed the code of the simple calculator. After it we need to develop the scientific calculator then we have two alternatives:

- (i) Make a new class.(Rewriting the class)
- (ii) Extending the class.

In the most cases second alternative will be a better choice. In this case we have to write the small piece of code. At the same time we won't have to spend much time in debugging and testing as the base class is already tested and is in use for a long period of time. When we have to inherit one class into another class then we have to use the “**extends**” keyword.

Syntax: class sub_class extends base_class



Class A (super class or base class or parent class).

Class B (child class or derived class or subclass).

The sub class will have all the features of base class in addition to the new features defined in the extended class. Java does not support the multiple-inheritance but its support multiple interface inheritance so sub class can extends only one base class.

Example 6.1:

```
1.      class A
2.      {
3.          int x;
4.          void setX(int x1)
5.          {
6.              x = x1;
7.          }
8.          void displayX()
9.          {
10.             System.out.println(x);
11.          }
12.     }
13.     class B extends A
14.     {
15.         int y;
16.         void setY(int y1)
17.         {
18.             y = y1;
19.         }
```

```
20.         void displayY()
21.         {
22.             System.out.println(y);
23.         }
24.     }
25.     class InheritTest1
26.     {
27.         public static void main(String args[])
28.         {
29.             A a1 = new A(); // 4 bytes
30.             a1.setX(5);
31.             a1.displayX();
32.             B b1 = new B(); // 8 bytes
33.             b1.setX(10);
34.             b1.setY(20);
35.             b1.displayX();
36.             b1.displayY();
37.         }
38.     }
```

Output:

5

10

20

Member Hiding:-

If a sub-class member has the same name (and same signature in case of methods) as that of a super-class member then it hides the super-class member.

Although both the members might be available in the sub-class but using member name we can only access sub-class member as it hides the member of the same name in the super-class.

Using keyword super

Whenever a subclass needs to refer to its immediate super class, it can do so by use of keyword super. super has two general uses:

1. Calling super class's constructor

A subclass can call a constructor method defined by its super class by use of the following form of super. **super (parameter-list);**

Here, parameter-list specifies any parameters needed by the constructor in the super class. In fact super() must always be the first statement executed inside a subclass's constructor.

2. Accessing a member of the super class that has been hidden by a member of a subclass.

The keyword super can be used to access the hidden members of the super as follows:

super.member;

Here, member can be either a method or a data member.

Method Overloading and Method Overriding:-

Method Overloading:-

When two or more methods having same name but different parameters then it is said to be a methods overloading. Method overloading is used when object are required to perform similar task but using different input parameters.

When we call a method, java matches up the method name first and then the number and type of parameters. To decide which one of the method definition is to called is known as polymorphism.

Method Overriding:-

We have seen that a method in a super class is inherited by its subclass and is used by the object created by the subclass. Method inheritance enables us to define and use method repeatedly in subclass without having to define the method again in subclass.

In the class hierarchy, when an instance method in a subclass has the same name signature as an instance method (non private) in its super class, then the method in the subclass is said to override the method in the super class.

When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the super class will be hidden.



Example 6.2

```
1.     class A
2.     {
3.         int x;
4.         void set(int x1)
5.         {
6.             x = x1;
7.         }
8.         void display()
9.         {
```

```

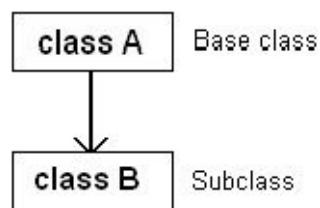
10.         System.out.println(x);
11.     }
12. }
13. class B extends A
14. {
15.     int y;
16.     void set(int x1,int y1) //set method of class B is not overriding set
17.         // method of class A as arguments are different
18.     {
19.         set(x1); //This will call set() of class A
20.         y = y1;
21.     }
22.     void display() //display method of class B is overriding display
23.         //method of class A
24.     {
25.         // display(); This will call display of class B
26.         super.display(); // This will call display of class A
27.         System.out.println(y);
28.     }
29. }
30. class InheritTest2
31. {
32.     public static void main(String args[])
33.     {
34.         B b1 = new B();
35.         b1.set(10,20);
36.         b1.display();
37.     }
38. }

```

Output:

Java support the following inheritance:

- [1] **Single inheritance.**
- [2] **Multilevel inheritance.**
- [3] **Hierarchical Inheritance.**

[1] Single inheritance.**Example 6.3:**

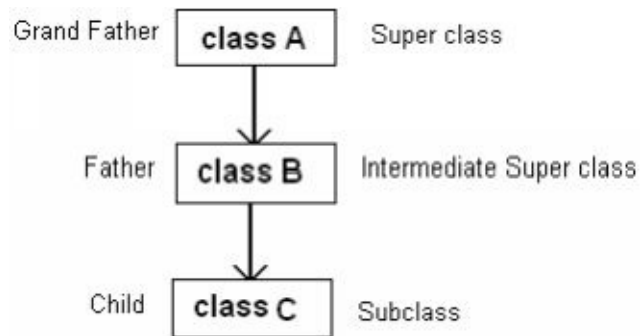
```
1.      class A
2.      {
3.          private int x;
4.          A()
5.          {
6.              x = 0;
7.          }
8.          A(int x1)
9.          {
10.             x = x1;
11.          }
12.         void display()
13.         {
14.             System.out.print(x);
15.         }
16.     }
```

```
17.     class B extends A
18.     {
19.         private int y;
20.         B()
21.         {
22.             super();
23.             //x=0; error as x is private
24.             y=0;
25.         }
26.         B(int x1, int y1)
27.         {
28.             super(x1);
29.             //x = x1; error as x is private
30.             y = y1;
31.         }
32.         void display()
33.         {
34.             super.display();
35.             System.out.print(y);
36.         }
37.     }
38.     class InheritTest3
39.     {
40.         public static void main(String args[])
41.         {
42.             B b1 = new B();
43.             b1.display();
44.             B b2 = new B(10,20);
45.             b2.display();
46.         }
47.     }
```

Output:

0 0 10 20

[2] Multilevel inheritance.



Example 6.4

```
1. class A
2. {
3.     private int x;
4.     A()
5.     {
6.         x = 0;
7.     }
8.     A(int x1)
9.     {
10.         x = x1;
11.     }
12.     void display()
```



```
13.         {
14.             System.out.println(x);
15.         }
16.     }
17.     class B extends A
18.     {
19.         private int y;
20.         B()
21.         {
22.             super();
23.             y=0;
24.         }
25.         B(int x1, int y1)
26.         {
27.             super(x1);
28.             y = y1;
29.         }
30.         void display()
31.         {
32.             super.display();
33.             System.out.println(y);
34.         }
35.     }
36.     class C extends B
37.     {
38.         private int z;
39.         C()
40.         {
41.             super();
42.             z=0;
43.         }
```

```
44.         C(int x1, int y1, int z1)
45.         {
46.             super(x1,y1);
47.             z = z1;
48.         }
49.         void display()
50.         {
51.             super.display();
52.             System.out.println(z);
53.         }
54.     }
55.     class InheritTest4
56.     {
57.         public static void main(String args[])
58.         {
59.             C c1 = new C();
60.             c1.display();
61.             C c2 = new C(10,20,30);
62.             c2.display();
63.         }
64.     }
```

Output:

0

0

0

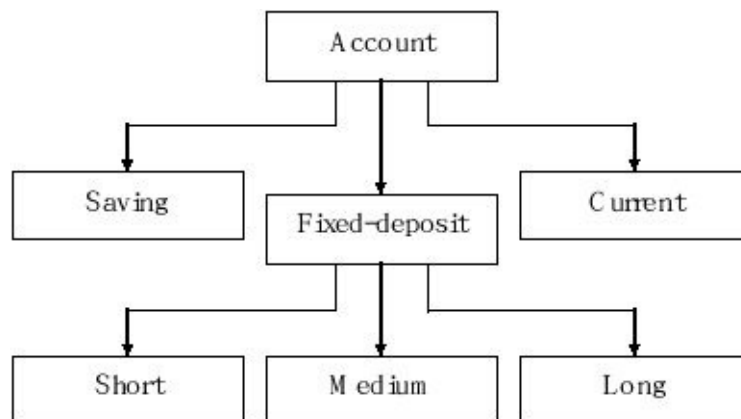
10

20

30



[3] Hierarchical inheritance.



Order of Constructor calling:-

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called? The answer is that in a class hierarchy, constructors are called in the order of derivation, from super class to subclass.

Further, since `super()` must be the first statement executed in a subclass's constructor; this order is the same whether or not `super()` is used. If `super()` is not used, then the default or parameter less constructor of each super class be executed.

Constructors are executed in order of derivation. Because a super class has no knowledge of any sub class, any initialization it needs to perform is separate and possibly pre-requisite to any initialization performed by the sub class. Therefore it must be executed first.

The keyword `super` is used subject to the following conditions.

super() constructor may only be called within a subclass constructor method.

The call to super class constructor must appear as the first statement within the subclass constructor.

The parameter in the super call must match the order and type of the arguments of the super class constructor.

Dynamic Method Binding -

(Dynamic Method Dispatch or Run Time Binding):-

Method overriding forms the basis for one of java's powerful concept Dynamic method dispatch is the mechanism by which call to an overridden instance method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how java implements run time polymorphism.

A super class reference variable can refer to a subclass objects. Java uses this fact to resolve calls to overridden method at run time. When an overridden method is called through a super class reference, java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.

Thus this determination is made at run time, when different types of objects are referred to different version of an overridden method will be called. In other words, it is the type of the object being referred to (not the type of reference) that determines which version of an overridden method will be executed.

Overridden methods allow Java to support run-time polymorphism. Polymorphism is essential for OOP for one reason. It allows a general class to specify methods that will be common to all of its derivatives, while allowing sup-classes to define the specific implementation of some or all of these methods.

By combining inheritance with overridden methods, a super class can define the general form of method that will be used by all of its sub-classes. The ability of existing code libraries to call methods on instances of new classes without recompiling while maintaining a clean abstract interface is a profoundly powerful tool.

Base class reference can take the sub class reference but subclass reference can not take the base class reference.



Example 6.5

```
1.      class Shape
2.      {
3.          void getdata()
4.          {
5.              System.out.println("getdata() of Shape");
6.          }
7.          void area()
8.          {
9.              System.out.println("area() of Shape");
10.         }
11.         void display()
12.         {
13.             System.out.println("display() of Shape");
14.         }
15.     }
16.     class Circle extends Shape
17.     {
18.         void getdata()
19.         {
20.             System.out.println("getdata() of Circle");
```

```
21.     }
22.     void area()
23.     {
24.         System.out.println("area() of Circle");
25.     }
26.     void display()
27.     {
28.         System.out.println("display() of Circle");
29.     }
30. }
31. class Rectangle extends Shape
32. {
33.     void getdata()
34.     {
35.         System.out.println("getdata() of Rectangle");
36.     }
37.     void area()
38.     {
39.         System.out.println("area() of Rectangle");
40.     }
41.     void display()
42.     {
43.         System.out.println("display() of Rectangle");
44.     }
45. }
46. class InheritTest5
47. {
48.     public static void main(String args[])
49.     {
50.         //super reference variable can accept sub class object
51.         //Dynamic binding
52.         Shape s[]={new Circle(), new Rectangle()};
```

```
53.         for(int i=0;i<s.length;i++)
54.         {
55.             s[i].getdata();
56.             s[i].area();
57.             s[i].display();
58.         }
59.     }
60. }
```

Output:

```
getdata() of Circle
area() of Circle
display() of Circle
getdata() of Rectangle
area() of Rectangle
display() of Rectangle
```

Abstract Class & Abstract Method:

We have seen that by making a method final we ensure that the method is not redefined in a subclass. Java allows us to do something that is exactly opposite to this. That is we can indicate that a method must always be redefined in a subclass, thus making overriding compulsory. This is done using the modifier keyword **abstract** in the method definition.

There are situations in which we will want to define a super class that declares the structure of a given abstraction without providing a complete implementation of every method.

That is, sometimes we will want to create a super class that only defines a generalized form that will be shared by all of its subclasses, leaving it to each sub class to fill in the details.

Such a class determines the nature of the methods that the sub classes must implement. One way this situation can occur is when a super class is unable to create a meaningful implementation for a method.

To declare an abstract method, use this general form:

```
abstract return_type name (parameter-list);
```

Nobody is present. Any class that contains one or more abstract methods must also be declared abstract. To declare a class abstract, we simply use the abstract keyword in front of the class keyword at the beginning of the declaration.

There can be no object of an abstract class. That is, an abstract class can not be directly instantiated with the new operator. Such objects would be useless, because an abstract class is not fully defined.

Also we cannot declare abstract constructors or abstract static methods. Any subclass of an abstract class must either implement all of the abstract methods in super class, or be itself declared abstract.

Example 6.6:

```
1.      abstract class Shape
2.      {
3.          abstract void getdata();
4.          abstract void area();
5.          abstract void display();
6.      }
7.      class Circle extends Shape
8.      {
9.          void getdata()
10.         {
11.             System.out.println("getdata() of Circle");
12.         }
13.         void area()
14.         {
```



```
15.         System.out.println("area() of Circle");
16.     }
17.     void display()
18.     {
19.         System.out.println("display() of Circle");
20.     }
21. }
22. class Rectangle extends Shape
23. {
24.     void getdata()
25.     {
26.         System.out.println("getdata() of Rectangle");
27.     }
28.     void area()
29.     {
30.         System.out.println("area() of Rectangle");
31.     }
32.     void display()
33.     {
34.         System.out.println("display() of Rectangle");
35.     }
36. }
37. class InheritTest6
38. {
39.     public static void main(String args[])
40.     {
41.         //super reference variable can accept sub class object
42.         //Dynamic binding
43.         Shape s[]={new Circle(), new Rectangle()};
44.         for(int i=0;i<s.length;i++)
45.         {
46.             s[i].getdata();
```

```

47.             s[i].area();
48.             s[i].display();
49.         }
50.     }
51. }

```

Output:

```

getdata() of Circle
area() of Circle
display() of Circle
getdata() of Rectangle
area() of Rectangle
display() of Rectangle

```

Final method:-

If we don't want a method to be overridden by a derived class method then we can define the base class method as final. Any attempt to override that method will cause a compile time error.

```

class A
{
    final void method1()
    {
        _____
    }
}
class B extends A
{
    void method1()           // error
    {
        _____
    }
}

```

```
}
```

Final class:-

Some times we may like to prevent a class being further sub-classed for security reason. A class that can not be sub classed is called a final class. This is achieved in java using the **keyword final** as follows:

```
final class A1
{
    -----
    -----
}
class B extends A           // error
{
    -----
    -----
}
```

Any attempt to inherit class A will cause an error and the compiler will not allow it. Declaring a class final prevents any unwanted extension to the class.

It also allows the compiler to perform some optimization when a method of a final class is invoked.

Theory Questions:

1. What is inheritance?
2. How many types inheritance are available in java.
3. What is difference between default constructor and parameterized constructor?
4. What is difference between final class and final method?
5. What is super class?
6. What is an abstract class?
7. What is difference between compile time and run time binding?
8. What is difference between method overloading and method overriding?



CHAPTER

∞ 7 ∞

(Object oriented programming)

Introduction-

Object oriented programming v/s procedural programming:-

All the programs consist of two elements: process (logic) and data. There can be two different approaches depending upon whether our main focus is on processing or data. The procedural programming languages (c, FORTRAN, PASCAL, COBOL etc.) focus on the processing part i.e. they give more importance to “what is happening” (process) in the system and less importance to “who is being affected” (data).

The **procedural approach** becomes less and less suitable as the programs become large and complex. The **object-oriented programming** languages were developed to overcome the limitations of the procedural programming languages. The object-oriented programming languages are relatively far less complex as compared to the similar programs within the procedural languages.

The object-oriented programs are organized around the data (i.e. objects) and a set of well defined interfaces (public methods) to that data.

Java is based on object oriented paradigm. Java is almost pure object oriented programming language. We have termed “almost” as Java also supports primitive data types due to performance reasons.

The C++ is not a pure object oriented language. C++ is an extension to C so it uses an approach, which is a mix of procedure-oriented approach and object-oriented approach.

The basic differences in the two approaches are summarized below:

- (i) The object-oriented programs are data centric while the programs written in the procedural languages are process centric.
- (ii) The object-oriented programs are organized around data (objects) so they model the real world objects in a better way.
- (iii) The degree of reusability and extensibility of the code is very high in case of object-oriented approach as compared to procedural approach. So code size is less.
- (iv) The object-oriented programs are easier to maintain, as they are relatively less complex and smaller in size.
- (v) The object-oriented programs are based on the bottom-up design methodology while the procedural programs are based on the top-down design methodology.

Basic concepts of OOP (Object-Oriented Programming):-

Some of the essential elements of the object oriented programming are mentioned below:

Abstraction

Objects and classes

Three OOP Principles

- Encapsulation

- Inheritance
- Polymorphism

Persistence

Genericity

Composition / Aggregation

1. Abstraction:-

The abstraction is one of the essential elements of any programming language including the procedural languages. The concept of built-in data type is also an abstraction. The feature of defining own data type using struct in C language further extends this abstraction.

The basic purpose of abstraction is to reduce complexity by hiding details. For example, a building is not thought to be a collection of building material but a well defined object having unique features and properties.

Abstraction can be defined as an act for identifying the essential properties and behavior of an object without going into details. The properties and behaviors of an object differentiate it from other objects of similar type and also help in classifying / grouping the objects.

The object oriented programming languages model abstraction using classes and objects.

2. Object and Classes:-

Object:-An object is a physical or abstract entity or thing, which can be distinguished from other objects of similar types. An object has three characteristics:

- (i) Identification.
- (ii) Properties. (Attributes)
- (iii) Behaviors. (Methods / Operations / Functions)

The object can also be thought of as an instance of class, where class is like a built-in data type and the object is the variable.

Class:-A class represents a category of objects by abstracting their properties and behaviors. A class can be thought of as a blueprint for creating objects. An object has the properties and behaviors defined by its class. The class can be thought of as a user-defined data type and object as a variable/instance of the data type represented by the class.

The properties/attributes of an object are defined by data members / fields in Java. A data member/field in a class is a variable that can hold data. The behaviors / operations of an object are defined using methods in Java. Both data members and methods are referred to as members of the class.

A class may also be thought of as a user-defined data type and an object as a variable of that data type. Once a class has defined we can create any number of objects belonging to that class.

3. Three OOP Principles:-

Encapsulation:- Encapsulation is the mechanism that binds code and data on which the code acts. Java classes support this as they allow us to define the code and data together. Encapsulation also helps in achieving data hiding by declaring some of the class members as private so that they cannot be accessed from the code that does not belong to the class.

Inheritance:- In object-oriented programming, Inheritance refers to the properties of a class being available to other classes called sub-classes or derived classes. A sub-class or derived class is one that is derived from an existing class. It inherits all the features of the existing class which is also referred as the base-class or super-class. So inheritance can be defined as the process of deriving a class from a super-class or a base-class. Inheriting a class does not introduce any changes in the base-class/super-class. The derived-class/sub-class has a larger set of properties and behaviors as compared to the base-class.

The major advantage of the inheritance is code reusability. If the base class is in use for a long time and there is a need to add some extra attributes and methods, we can do so by deriving another class. The derived class will be able to use code of the base class without debugging as it is in use for a long time.

Class Hierarchy:-All the classes derived from a common base class belong to a family and form a class hierarchy. The class hierarchy can be compared with a tree structure where one base class is at the root and does not have a super-class.

All other classes are either derived from the class at the root of the hierarchy or from some other class, which is derived from the root class directly or indirectly. More features are added as we go down the tree. The classes that are represented by the leaf nodes do not have any sub-classes. The following example shows some class hierarchies.

Polymorphism:-

Polymorphism is a feature that allows same interface to be used for a general class of actions. Most of the object-oriented languages use polymorphism in the following situations.

- ✚ Operator Overloading.
- ✚ Method Overloading.
- ✚ Method Overriding.

Operator Overloading:-

Most of the languages use this form of polymorphism for the built-in operations. For example, all the arithmetic operators in C/C++ or Java can be used with many types of operands (int, long, float, double etc.). So same addition operators can be used to add two integers as well as to add two floating point numbers.

The C++ allows the user to overload the built in operators. For example, we can overload the arithmetic operators to handle the complex numbers also. **Although Java uses operator overloading for built-in operators but does not allow the user to overload the operators.**

Method Overloading:-

This feature allows us to write more than one methods with the same name. Both C++ and Java have this feature. The methods (function in C++) with same name are differentiated based on the parameters (arguments). The overloaded methods must either have different number of parameters or the types of the parameters must differ if their number is same.

If the call to an overloaded method can be resolved at compile time i.e. if the compiler can decide which of the overloaded method will be called then this is called static binding, which is an example of compile time polymorphism.

If the call to an overloaded method can not be resolved at compile time i.e. if the compiler can not decide which of the overloaded method will be called then this is called dynamic binding, which is an example of run-time polymorphism.

In general Java resolves calls to overloaded methods at run-time but there are many situations where the calls to overloaded methods are resolved at compile-time.

Method Overriding :-

The sub-class can define a method with the same name as in the super class, and the same number and type of parameters. This is called method overriding.

The compiler can not resolve calls to overridden methods. Java normally uses dynamic binding to resolve calls to overridden methods i.e. the decision takes place at run-time.

4. Persistence:-

Some object-oriented languages allow us to store / retrieve the state of a program to / from a persistence storage media i.e. a permanent storage media like secondary storage. This is called persistence.

Java allows us to store any object on the secondary storage and to retrieve it later on. If we attempt to store an object at the top of an object graph, all of the other referenced objects are recursively located and saved. Similarly when the object is retrieved later on, all of the objects and their references i.e. the entire object graph is correctly created in the main memory.

For example, it is possible to save an entire tree structure by just saving the root of the tree. At a later stage it is possible to recreate the entire tree structure in the memory by just saving the root of the tree. The C++ does not support this feature.

5. Genericity:-

The concept of defining an algorithm once, independently of any specific type of data, and then applying that algorithm to a wide variety of data types without any additional effort is called Genericity. C++ supports this feature using templates. Java also supports this feature through Object class, which is at the top of any class hierarchy in Java.

For example, using this feature, we can implement a generic data type stack so that it is possible to store element of any type in the stack. This feature increases the degree of reusability to a large extent.

6. Composition / Aggregation

An object might be made up of other objects. Such an object is called Composite or Aggregate object. For example, it is appropriate if an object of class vehicle is defined as a composite object made up of objects like Engine, Body, Axle, Seats etc.

Inheritance v/s Composition:-

There are two basic mechanisms for deriving new classes from the existing ones: Inheritance and composition. The class Bus can be derived by inheriting properties of class vehicle. Here Bus is a kind of vehicle which has some additional properties / behaviors beside the properties and behaviors which are common for all the vehicles.

In other words class Bus has is-a relationship with the class vehicle as we can say that Bus is a vehicle.

The class vehicle itself might be derived from many other classes using composition. For example, an object of class vehicle might be composed of objects belonging to classes like Engine, Gear Box, Seats, Driver's Seat Body, and Steering Wheel etc.

The derived class in this case has whole-part relationship with the classes representing parts of the composite object. We can not say that vehicle is an Engine or Vehicle is a Gear Box as Vehicle is made up of a number of parts.

Super classes and Subclasses

Often an object of one class “is an” object of another class as well. A rectangle certainly is a quadrilateral (as are squares, parallelograms and trapezoids). Thus, class Rectangle can be said to inherit from class Quadrilateral. In this context, class Quadrilateral is a superclass, and class Rectangle is a subclass.

A rectangle is a specific type of quadrilateral, but it is incorrect to claim that a quadrilateral is a rectangle (the quadrilateral could be a parallelogram). Figure shows several simple inheritance examples of superclasses and potential subclasses. Inheritance normally produces subclasses with more features than their superclasses, so the terms superclass and subclass can be confusing.

There is another way, however, to view these terms that makes perfectly good sense. Because every subclass object “is an” object of its superclass, and because one superclass can have many subclasses, the set of objects represented by a superclass is normally larger than the set of objects represented by any of that super class’s subclasses.

For example, the superclass Vehicle represents in a generic manner all vehicles, such as cars, trucks, boats, bicycles and so on. However, subclass Car represents only a small subset of all the Vehicles in the world. Inheritance relationships form tree-like hierarchical structures.

A superclass exists in a hierarchical relationship with its subclasses. A class can certainly exist by itself, but it is when a class is used with the mechanism of inheritance that the class becomes either a superclass that supplies attributes and behaviours to other classes or a subclass that inherits those attributes and behaviours. Frequently, one class is both a subclass and a superclass.

Superclass	Subclasses
Student	Graduate student Under Graduate student
Shape	Circle Triangle Rectangle
Loan	Car Loan Home Loan Education Loan
Employee	Faculty Member Staff Member
Account	Checking Account Saving Account



CHAPTER

∞ 8 ∞

(PACKAGES)

Introduction-

When we work on a project we have to break our program in several classes. To organize our classes we use Packages. The package is both a naming and visibility control mechanism. Package is a collection of classes and interfaces which are interrelated. With the help of package it is possible to give same names to more than one class provided they are defined in different package.

A single package can't contain two classes with same name.

PROGRAMS are organized as sets of packages. Each package has its own set of sub-packages, which helps to prevent name conflicts. A top level package is accessible outside the package only if it is declared as public. The naming structure for packages is hierarchical. The members of a package are class and interface types and sub-packages. A package in java is an encapsulation mechanism that can be used to group related class, interface and sub-packages.

For small programs, a package can be unnamed or have a simple name, but if code is to be widely distributed, unique package names should be chosen.

This can prevent the conflicts that would otherwise occur if two development groups happened to pick the same package name and these packages were later to be used in a single program. If we do not specify a package for a java class then that class will be the part of the default package of Java called as unnamed package. But this requires that every class must have a unique name to avoid collision.

Example of the unnamed package.

```
class A
{
    -----
    -----
}
class A_demo
{
    public static void main(String args[])
    {
        -----
        -----
    }
}
```

How to define a package

To create a package, include a package command as the first statement in a Java source file. Any class declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If we omit the package statement, the class names are put into the default package, which has no name and is called as un-named package.

The package statement has the following syntax:

package package_name;

Java uses file system directories to store packages. Remember that the case is significant, and directory name must match the package name exactly. More that one file can include the same package statement.

Syntax:

```
package p1;           //package declaration
class A               //class declaration
{
    -----          //body of class A
    -----
}
```

package is a keyword. p1 is the package name in which class A is to be stored.

Example 8.1:

```
1. package p1;
2. class A
3.     {
4.         public static void main(String args[])
5.             {
6.                 System.out.println("First Package program.");
7.             }
8.     }
```

We have assumed that the current folder is (c:\javaprg\p1>)

C:\javaprg\p1>javac A.java

C:\javaprg\p1>java p1.A

Output:

First Package Program

Note:-If we compile the above program with -d "javac -d . A.java" in "c:\javaprg>" then the folder p1 will be created automatically and the .class file of the above program will be saved in this folder. But if we do not compile with -d option then it is our duty to create the p1 folder and place the .class file in that folder. (.) directs the compiler to create the p1 folder in the current folder. We can also type the complete path if we want to create the folder in some other folder as **javac -d c:\javaprg A.java**

Importing packages:

All the inbuilt classes of Java are stored in some named packages; no class is stored in the unnamed default package. Since classes within packages must be fully qualified with their package name or names. It could become tedious to type in the long dot separated package path name for every class we want to use. For this reason, java includes the import statement to bring certain classes, or entire packages, in to visibility. Once imported, a class can be referred to directly, using only its name.

The import statement is a convenience to the programmer and is not technically needed to write a complete java program. If we are going to refer to a few dozen classes in our application, then the import statement will save a lot of typing.

In a java source file, import statement occurs immediately following the package statement (if it exists) and before any class definition. This is the general form of the import statement.

```
import pkg1[.pkg2].classname;
```

```
import pkg1[.pkg2].*;
```

Example 8.2

```
1 //Program to print any 10 random numbers using class Random of package
2.      //java.util without using import statement
3.      class RandomTest1
4.      {
5.          public static void main(String args[])
6.          {
7.              java.util.Random r1 = new java.util.Random();
8.              for(int i=1;i<=10;i++)
9.              {
10.                 System.out.println(Math.abs(r1.nextInt()));
11.             }
12.         }
13.     }
```

Output:

1588659525
481996564
210875677
386971039
360473512
1922641171
393055462
889442536
1836112189
320882863

Example 8.3

```
1. //Program to print any 10 random numbers using class Random of package
2.     //java.util using import statement
3.     import java.util.Random;
4.     class RandomTest2
5.     {
6.         public static void main(String args[])
7.         {
8.             Random r1 = new Random();
9.             for(int i=1;i<=10;i++)
10.            {
11.                System.out.println(Math.abs(r1.nextInt()));
12.            }
13.        }
14.    }
```

Output:

686722541

1154462827
452959606
1235134497
1421490552
1065881855
424893046
731473166
1217844570
437181304

CLASSPATH:

If we run the above program from any other folder other than the current folder then it will not run. By default the Java run-time system uses the current working directory as its starting point.

Thus if our package is in the current directory, or a subdirectory of the current directory, it will be found. But to run the above program from any other folder we have to set the class path using command `SET CLASSPATH=.;C:\JAVAPRG` on command prompt. CLASSPATH is a environment variable. We have to type this command every time when we start the computer for the first time. But if we put this command in **MyComputer properties advanced Environment variables** then there is no need to type this command again & again.

We can also run the program without setting the CLASSPATH environment variable. To do this type

```
java -classpath c:\javaprg p1.A
```

Note4:-We can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a dot. The general form of a multi-leveled package statement is:

```
package package_name1[.package_name2][.package_name3];
```

A package hierarchy must be reflected in the file system of Java development system. For example package **java.awt.Image** need to be stored in **java\awt\image** (WINDOWS) or **java/awt/image** (UNIX) or **java:swt:image** (MACINTOSH) file system.

A package hierarchy represents an organization of the java classes and interfaces. It does not represent the source code organization of the classes and interfaces.

Each java source file (also called compilation unit) can contain zero or more definition of classes and interfaces, but the compiler produces a separate class file containing the java byte-code for each of them. A class or interface can indicate that its java-byte code be placed in a particular package, using a package declaration.

The package java has sub-packages awt, applet, io, lang, net, and util, but no classes or interface.

The package java.awt has a sub-package named image, as well as a number of classes and interfaces.

Because the package java.awt has a sub-package image, it cannot contain a declaration of a class or interface type named image.

If the fully qualified name of a package is P , and Q is a sub-package of P , then $P.Q$ is the fully qualified name of the sub-package.

If there is a package named mouse and a member class Button in that package (which then might be referred to as mouse.Button), then there cannot be any package with the fully qualified name mouse.Button or mouse.Button.Click.

At most one package statement can appear in a source file, and it must be the first statement in the Java source file.

Visibility of class member:

Class and package are both means of encapsulating and containing the name space and scope of variable and methods. Packages act as containers for classes and other subordinate packages. Class act as a container for data and method code.

The class is java's smallest unit of abstraction. Because of the interplay between the classes and packages, Java addresses five categories of visibility of class members.

1. Visibility within the class.
2. Visibility in subclass in the same packages.
3. Visibility in non subclass in the same packages.
4. Visibility in subclasses in different packages.
5. Visibility in subclasses that are neither in the same package nor are subclass.

Note1:-

Top level class and interface has only two possible access **default** and **public**.

Note2:-If class is declared **public** then it is accessible by using other code.

Note3:-If class has **default** access then it can only be accessed by other code within the same package.

Note4:-The member visibility has meaning only if the class is visible. If visibility modifier of the class is default then even public members of the class will be visible only within the package.

Access specifiers:

access modifier →	public	protected	default	private
access location ↓				
Same class	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	No
Other class in same package	Yes	Yes	Yes	No
sub class in other package	Yes	Yes	No	No
Non subclass in other package	Yes	No	No	No

JAR files: (Java Archive Files)

To compress a file we use some program such as winzip in the same way java's JAR feature can also be used to compress the entire hierarchy of a Java package. It provides a better facility for installation.

There is no need to create the file structure at the customer location. We have to just use JAR file and this file will restore the entire file structure. All the package, sub-package and class files will be created automatically from the JAR file.

JAR technology makes it much easier to deliver and install software. Also the element in a JAR file are compressed, which make downloading a JAR file much faster than separately downloading several uncompressed file. This allows a consumer to be sure that these elements were produced by a specific organization or individual.

There are following option are available for the JAR files:

Option	Description
c	A new archive is to be created.
C	Change directories during command execution.
f	First element of the file list is the name of the archive that is to be created.
i	Index information should be provided
m	The second element in the file list is the name of the external manifest file.
M	Manifest file not created
t	The archive contents should be tabulated
u	Update existing JAR file.
v	Verbose output should be provided by the utility as it execute.
x	Files are to be extracted from the archive
o	Do not use compression

Creating a JAR file:

```
c:\javaprg> jar -cf Myjar.jar pack1
```

This will create a JAR file “Myjar.jar” which will contain all the files of package pack1 in compressed form

Extract JAR files:

This command used for the extract the jar files.(such as unzip)

```
c:\javaprg> jar -xf Myjar.jar
```

Tabulating the Contents of a jar file:

The following commands list the contents of Myjar1.jar.

```
c:\javaprg> jar -tf Myjar.jar
```

Updating an existing JAR file:

The following command is useful for the update of the jar file.

```
c:\javaprg>jar -uf Myjar.jar pack1
```

Creating an executable JAR file:

```
c:\javaprg>jar -cmf mainClass Myjar.jar pack1
```

where mainClass is a text file which contains “Main-Class: pack1.pack2.pack3.A” saved in javaprg folder.

Execute a jar file:

```
c:\javaprg>java -jar Myjar.jar
```

Theory question's:

1. What do we understand about the packages?
2. Explain all the modifiers and its scope in different packages and sub packages with subclass and other class.
3. What is the use of JAR files?
4. What is the use of the import statement?

CRACKING THE PROGRAMMING INTERVIEW



2000+ JAVA INTERVIEW QUE. / ANS.
500+ Non-Technical Interview Que. / Ans.

- Harry Anonymous Hactivist.

[Search This book on Amazon.com with ISBN- 978-1500730413.](#)



CHAPTER

∞ 9 ∞

(Interface)

Introduction-

An interface is basically kind of class. Like classes, interface contain methods and variables but with a major difference. The difference is that interface defined only abstract methods and final variables (Constants). This means that interface do not write any code to implements these methods and data fields contain only constants.

Therefore, it is the responsibility of the class to implement an interface defining the code for implementation of these methods. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without a body. In practice, this means that we can define interfaces, which do not make assumptions about how they are implemented. The purpose of the interface is to separate a class's interface from its implementation. Interface just defines what a class must do without saying anything about the implementation. Interfaces define only method signatures and they do not have any instance variables.

A *nested interface* is any interface whose declaration occurs within the body of another class or interface. A *top-level interface* is an interface that is not a nested interface. The syntax for defining an interface is very similar to that for defining a class. The general form of an interface definition is:

```
interface interface_name
```

```
{  
    variable declarations;  
    methods declarations;  
}
```

Here, interface is the keyword and interface_name is any valid java identifier (just like class name).

Note: When we define an interface then by default it is public and abstract.

Interface Modifiers:

The only modifier that can be used with the top-level interfaces are abstract and public. Even if we do not use abstract modifier, the interface is implicitly declared to be abstract so use of modifier abstract is not required. The visibility of a top-level interface can be either package or public just like top-level class. If no visibility modifier is used then the visibility of the interface is assumed to be package.

Data Variable Member Declarations:

Only constants can be defined in the interface. All the variables in an interface are implicitly public, final and static meaning they cannot be changed by the implementing class. They must also be initialized with constant value.

public static final type variable_name=value;

The variable declared in an interface is public, static and final. It means it is a static variable with public access. The variable is also declared final, which means that it must be assigned a value at the time of declaration, which can not be modified later on. The keyword final is like const in C/C++.

Methods Declarations:

Methods declaration will contain only a list of methods without anybody statements. They end with a semicolon after the parameter list. They are essentially, abstract methods and there can be no default implementation of any method specified within an interface.

All methods in the interface are always public and abstract modifiers by default. We cannot declare static methods in interfaces.

```
public abstract return_type method_name(parameter_list);
```

Here is an example of an interface definition that contains two variables and one methods.

```
interface item
{
    static final int code =1001;
    static final String name = “Matrix”;
    abstract void display( );
}
```

Note:-Code for the methods is not included in the interface and methods declaration simply ends with a semicolon.

Another example of an interface:

```
interface area
{
    float pi = 3.142f;
    float compute (float x, float y);
    void show( );
}
```

Extending interface:

Like classes, interface can also be extended i.e. an interface can be sub-interfaced from other interface. The new interface will inherit all the members of the sub interface in the manner similar to subclass. This is achieved using the keyword extends as shown below.

```
interface interface_1
```

```

{
    int code = 1001;
    String name = "Matrix";
}
interface interface_2 extends interface_1
{
    void display();
}

```

The interface_2 would inherit both the constant code and name in to it.

Note:-

That the variables name and codes are declared like a simple variable. It is allowed here but all the variables in the interface are treated as constant, public & static although the keywords public, static and final are not here.

We can combine several interfaces together in to a single interface. Following declarations are valid.

```

interface interface_1
{
    int code = 1001;
    String name = "Matrix";
}
interface interface_2
{
    void display();
}
interface interface_3 extends interface_1, interface_2
{
    -----
    -----
}

```

Note:-

Always remember that an interface can not extend classes. This would violate the rule that an interface can have only abstract methods and constants.

Implementing Interface:

Interfaces are used as “super-classes” whose properties are inherited by classes. It is therefore necessary to create a class that inherits the given interface. Once interface has been defined, one or more classes can implement that interface.

To implement an interface, include the implements clause in a class definition, and then create the methods declared by the interface.

The implementing class must provide body of all the methods in all the interfaces otherwise it must be declared as abstract.

The implementing class may also extend a class and can implement multiple interfaces.

```
modifier class<class-name>implements<interface-name>
{
    body of the class
}
```

Syntax of an interface implements in the class

```
interface interface_name
{
    -----
}
class class_name implements interface_name
```

```

{
    Body of class
}

```

Java does not support **multiple inheritance** but it support **multiple interface inheritance** i.e. a class can implement more than one interfaces as show below:

```

modifiers class <class-name> implements <interface-1>,<interface-2>,
<interface-3>, ....<interface-n>
{
    body of class
}

```

Note:-

When we implement an interface method, it must be declared as public. The implementing an interface method is like over-riding so we can not decrease the visibility.

Example 9.1

```

1.     interface interface_1
2.     {
3.         void show(int x);
4.     }
5.     class class_1 implements interface_1
6.     {
7.         public void show(int x);
8.         {
9.             -----
10.        }
11.    }

```

Example 9.2 of abstract class

```

1.         interface common

```

```

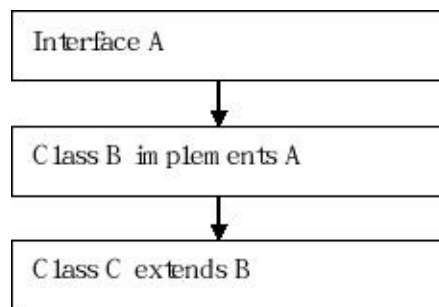
2.      {
3.          void push(int x);
4.          int pop ( );
5.      }
6.      abstract class stack implements common
7.      {
8.          void disp()
9.          {
10.             -----
11.          }
12.      }

```

Here the class stack does not implements the push() and pop() methods. So we must declare stack class as a abstract class this technique is called to be a ***partial implementation***.

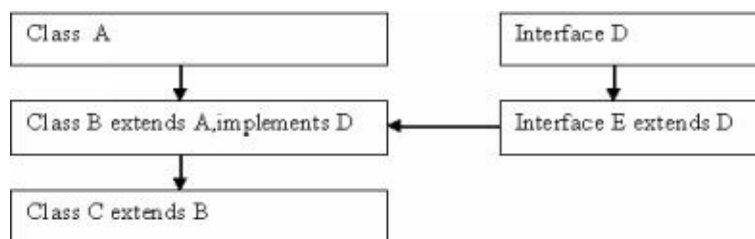
Various form of interface inheritance:-

(a)



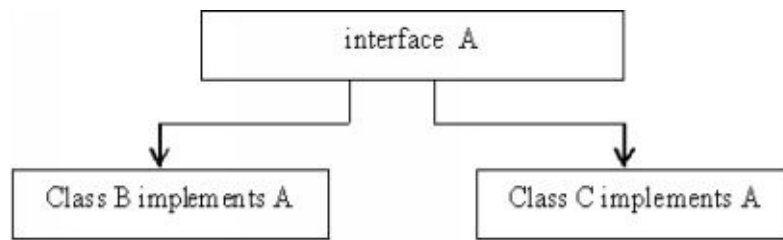
(single level interface inheritance)

(b)



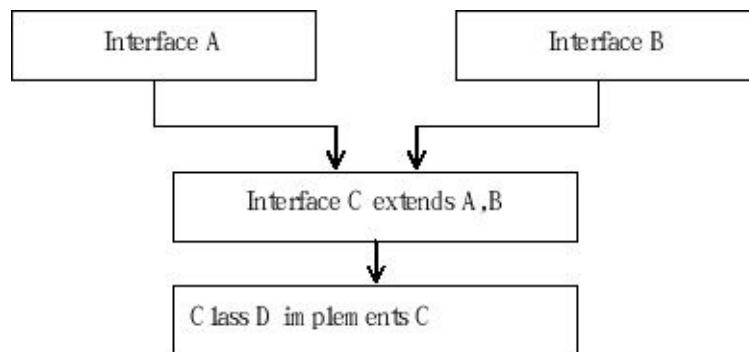
(multilevel interface inheritance)

(c)



(hierarchical interface inheritance)

(d)



(hybrid interface inheritance)

Accessing Implementing Class Objects Through Interface Reference:

We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When we call a method through one of these reference, the correct version will be called base on the actual instance of the interface being referred to.

This is one of the important feature of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be create later than the code, which calls methods on them. The calling code can dispatch through an interface without having to know anything about the “callee”. This process is similar to using a super class reference to access a sub class object.

Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, we should be careful not to use interface casually in performance-critical code.

Example 9.3

```
1.         interface Math_function
2.         {
3.             void display(int x);
4.         }
5.         class Sqrt implements Math_function
6.         {
7.             public void display(int x)
8.             {
9.                 System.out.println("sqrt of x =" + Math.sqrt(x));
10.            }
11.        }
12.        class Log implements Math_function
13.        {
14.            public void display(int x)
15.            {
16.                System.out.println("log of x = " + Math.log(x));
17.            }
18.        }
19.        class Math_Demo
20.        {
21.            public static void main(String args[])
22.            {
23.                Math_function f1 = new Sqrt();
24.                Math_function f2 = new Log();
25.                f1.display(27);
26.                f2.display(3);
27.                Math_function f3;
28.                f3=f1;    //polymorphism (assign the reference)
29.                f3.display(30);
30.                f3=f2;    //polymorphism (assign the reference)
```

```
31.                f3.display(5);
32.                }
33.        }
```

Output:

```
sqrt of 27 = 5.196152422706632
log of 3 = 1.0986122886681096
sqrt of 30 = 5.477225575051661
log of 5 = 1.6094379124341003
```

Characteristics of Interface:

Interface is a keyword.

Interface can be declared as abstract but it is abstract default so there is no need to add abstract keyword.

All variables are always public static & final.

All methods of interface must be implemented in the class.

All methods implemented by the implementing class must be public.

Interface includes declaration of the methods only.

Interface methods can not be declared as static. They are always declared as an instance methods.

A class can neither narrow the accessibility of an interface method nor specify new exceptions in method's throws clause; as attempting to do so would amount to altering the interfaces contract, which is illegal. The criteria for overriding methods also apply when implementing interface methods.

All methods need to be defined as public in the implementing class.

Partial implements are allowed here using by abstract keywords.

Regardless of how many interfaces a class implements directly or indirectly, it only provides a single implementations of a method that might have multiple declarations in the interfaces.

Method prototype declarations can also be overloaded as in the case of classes.

An interface constant can be accessed by any client (a class or interface) using its fully qualified name, regardless of whether the client extends or implements its interface. However, if a client is a class that implements this interface or an interface that extends this interface, then the client can also access such constants directly without using the fully qualified name. Such a client inherits the interface constants.

In the case of multiple inheritance of interface constants any name conflicts can be resolved using fully qualified names for the constants involved.

Interface Variables:

All variables in the interface are always public static and final because they are common part for the access of the data types and methods for the implementing.

Memory Management of the Interface Variable:

Static variables always get the memory when our program is to be loaded they take the separate memory and do not share with the any object's memory. They are also called as constant variables. They can be accessed directly using interface name and the {.} dot.

Example 9.4

```
1.          interface Static_Var
2.          {
3.              int sun=1;
4.              int mon=2;
5.              int tues=3;
6.              int wed=4;
7.              int thurs=5;
```

```
8.         int fri=6;
9.         int sat=7;
10.        }
11.    class Week_Day implements Static_Var
12.    {
13.        public static void main(String args[])
14.        {
15.            System.out.println(sun);
16.            System.out.println(mon);
17.            System.out.println(tues);
18.            System.out.println(wed);
19.            System.out.println(thurs);
20.            System.out.println(fri);
21.            System.out.println(sat);
22.        }
23.    }
```

or

```
11.    class Week_Day
12.    {
13.        public static void main(String args[])
14.        {
15.            System.out.println(Static_Var.sun);
16.            System.out.println(Static_Var.mon);
17.            System.out.println(Static_Var.tues);
18.            System.out.println(Static_Var.wed);
19.            System.out.println(Static_Var.thurs);
20.            System.out.println(Static_Var.fri);
21.            System.out.println(Static_Var.sat);
22.        }
23.    }
```

Output: 1

2

3

4

5

6

7

Example 9.5

```
1.         interface Two_Methods
2.             {
3.                 void m1( );
4.                 void m2( );
5.             }
6.         class Three_Methods implements Two_methods
7.             {
8.                 public void m1( )
9.                 {
10.                    body of the m1;
11.                }
12.                public void m2( )
13.                {
14.                    body of the m2;
15.                }
16.                public void m3( )
17.                {
18.                    body of the m3;
19.                }
20.            }
21.         class Methods_Demo
22.             {
23.                 public static void main(String args[])
24.                 {
25.                    Two_methods tm1 = new Three_methods();
26.                    tm1.m1();
27.                    tm1.m2();
28.                    //tm1.m3();          error
29.                    Three_Methods tm2 = new Three_Methods();
```

```
30.             tm3.m3();
31.         }
32.     }
```

Note:- Here we can not call the m3() using interface reference because this methods is not declared in the interface. So interface reference can access those methods which are declared in the self block. Otherwise we can call the same classes objects or extended classes objects.

Example 9.6 of two interfaces implemented in the one class

```
1.     interface Two_Methods
2.     {
3.         void m1();
4.         void m2( );
5.     }
6.     interface One_Method extends Two_Methods
7.     {
8.         void m3();
9.     }
10.    class Three_Methods implements One_methods,
11.    {
12.        public void m1( )
13.        {
14.            body of the m1;
15.        }
16.        public void m2( )
17.        {
18.            body of the m2;
19.        }
20.        public void m3( )
21.        {
22.            body of the m3;
23.        }
24.    }
```

```

25.     class Methods_Demo
26.     {
27.         public static void main(String args[])
28.         {
29.             Three_methods tm1 = new Three_methods();
30.             tm1.m1();
31.             tm1.m2();
32.             tm1.m3();
33.         }
34.     }

```

Non Interface Methods: When any class implements the methods of the interface and also defines additional methods in the class which are not declared in the interface, then we can say that the additional methods are non interface methods. Using reference variable of interface we can't call these additional methods.

Example9.7

```

1.     interface Two_Methods
2.     {
3.         void push(int x);
4.         int pop();
5.     }
6.     class Stack implements Two_Methods
7.     {
8.         int arr[] = new int[5];
9.         int top= -1;
10.        public void push(int item)
11.        {
12.            if(top == 4)
13.            {
14.                System.out.println("Overflow");
15.                return;

```

```

16.         }
17.         top++;
18.         arr[top]=item;
19.     }
20.     public int pop()
21.     {
22.         if(top == -1)
23.         {
24.             System.out.println("Underflow");
25.             return -1;
26.         }
27.         int item = arr[top];
28.         top--;
29.         return (item);
30.     }
31.     void disp()
32.     {
33.         for(int i=top;i>=0;i--)
34.         {
35.             System.out.println(arr[i]);
36.         }
37.     }
38. }
39. class Stack_Demo
40. {
41.     public static void main(String args[])
42.     {
43.         Stack s1=new Stack();
44.         s1.pop();
45.         for(int i=0;i<5;i++)
46.             s1.push(i+100);
47.         s1.push(600);

```



```

48.         System.out.println("--Displaying All Item--");
49.         s1.disp(); //non interface methods
50.         System.out.println("--Removing All the Items--");
51.         for(int i=0;i<5;i++)
52.             {
53.         System.out.println(s1.pop()); //-1 for underflow
54.             }
55.         Two_Methods t1=new Stack();
56.         t1.push(10);
57.         System.out.println(t1.pop());
58.         //t1.disp();          error
59.     }
60. }

```

Output:

```

Underflow
Overflow
--Display All Item--
104
103
102
101
100
--Remove All Items--
104
103
102
101
100
10

```





CHAPTER

∞ 10 ∞

(String and StringBuffer)

Introduction-

In java Strings are class objects and implemented using two classes, namely **String** and **StringBuffer**. Java String, as compared to C strings are more reliable and predictable. This is basically due to C's lack of bound checking. A java String is not a character array and is not NULL terminated. In general, Java does not allow operators to be applied to String objects.

The one exception to this rule is + & += operator, which concatenates two strings producing a String object as a result. In Java character array is not treated as string. In Java String class is defined to do all string operations. The class String and StringBuffer are part of the java.lang package.

String class:

```
String s1 = new String("matrix");
String s2 = "matrix";
System.out.println(s1);
System.out.println(s1.length);
System.out.println("matrix".length);
```

The String has the following characteristics in java:

[i] String is an object in java:

String represent a sequence of characters, But unlike many other language that implements String as character arrays, java implements string as objects of type string.

String manipulation is the most common part of many java programs. Implementing strings as built in objects allows java to provide a full complement of features that make string handling convenient. Also string objects can be constructed a number of ways, making it easy to obtain a string when needed.

[ii] String is immutable:

String are immutable i.e. we cannot change the string after creation. However a variable declared as a String reference can be change to point some other string object at any time.

We can still perform all type of string operation. Each time we need an altered version of an existing string, a new string object is created that contains the modifications.

The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones.

For those cases in which a modified string is desired, there is a companion class called StringBuffer, whose objects contain strings that can be modified after they are created.

[iii] String class is final:

Both String and StringBuffer classes are final. It means we can't extend class String in any other class. This allows certain optimization that increase performance of common string operations.

Constructors & Methods of String class:

1 public String();

Initializes a newly created String object so that it represents an empty character sequence.

String s1 = new String();

2 public String(String str);

Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.

String s1 = new String("Matrix");

String s2 = new String(s1);

The string s2 is a copy of string s1. Although contents are same but s1 and s2 points to different string objects.

3 public String(char ch[]);

Allocated a new String so that it represents the sequence of characters currently contained in character array argument.

char ch[] = {'a', 'b', 'c'};

String s1 = new String(ch); //s1 will hold "abc"

4 public String(char value[], int offset, int count);

Allocates a new String that contains characters from a sub-array of the character array argument.

char ch[] = {'a', 'b', 'c', 'd', 'e', 'f'};

String s1 = new String(ch,2,3); //s1 will hold "cde"

5 public String(int value[], int offset, int count);

Allocates a new String that contains characters from a sub-array of the integer array argument containing Unicode of characters.

int a[] = {97, 98, 99, 100, 101, 102};

String s1 = new String(a,2,3); //s1 will hold "cde"

6 public String(byte b[], int offset, int count);

Constructs a new String by decoding the specified sub-array of bytes using the platform's default char set.

```
byte b[ ] = {97, 98, 99, 100, 101, 102};
```

```
String s1 = new String(b,2,3); //s1 will hold "cde"
```

7 public String(byte[]);

Constructs a new String by decoding the specified array of bytes using the platform's default char set.

```
byte b[ ] = {97, 98, 99};
```

```
String s1 = new String(b); //s1 will hold "abc"
```

Even though Java's char type used 16 bits to represent the Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the String class provides constructors that initialize a string when given a byte array. In each of the above constructors the byte to character conversion is done by using the default character encoding of the platform.

8 public String(StringBuffer);

9 public int length();

This will display number of characters of the string.

10 public boolean isEmpty();

This will display true if the string is empty otherwise false.

```
String s1 = new String( );
```

```
System.out.println(s1.isEmpty());
```

11 public char charAt(int index);

Returns the char value at the specified index. An index ranges from 0 to length()-1. The first char value of the sequence is at index 0, the next at index 1, and so on, as for array indexing. Throws **StringIndexOutOfBoundsException** if invalid index is specified.

```
String s1 = "matrix";
```

```
System.out.println(s1.charAt(0)); // will print "m"
```

12 public void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin);

Copies characters from this string into the destination character array. The first character to be copied is at index `srcBegin`, the last character to be copied is at index `srcEnd-1` (thus the total number of characters to be copied is `second-srcBegin`). The characters are copied into the subarray of `dst` starting at index `dstBegin` and ending at index: `dstBegin+(srcEnd – srcBegin)–1`. This method may throw `ArrayIndexOutOfBoundsException` and `StringIndexOutOfBoundsException`.

13 `public void getBytes(int srcBegin, int serEnd, byte dst[], int dstBegin);`

14 `public byte[] getBytes();`

Encodes this `String` into a sequence of bytes using the platform's default charset, storing the result into a new byte array. This method is most useful when we are exporting a string value into an environment that does not support 16-bit Unicode character. For example, most Internet protocols and text file formats use 8-bit ASCII for all text interchange

15 `public boolean equals(Object);`

Compares the invoking string with the specified object. The result is true if and only if the argument is not null and is a `String` object that represents the same sequence of characters as the invoking string.

16 `public boolean equalsIgnoreCase(String);`

Compares the invoking `String` with the `anotherString`, ignoring case considerations.

17 `public int compareTo(String);`

Compares the invoking `String` with the string passed as argument, lexicographically.

Return value	Meaning
---------------------	----------------

<0	The invoking string is less than <code>str</code>
----	---

>0	The invoking string is greater than <code>str</code>
----	--

0	The two strings are equals
---	----------------------------

18 `public int compareToIgnoreCase(String);`

Compares two strings lexicographically, ignoring case differences.

19 `public boolean regionMatches(int startIndex, String str2, int str2StartIndex, int len);`

Tests if two string regions are equal.

Public boolean regionMatches(Boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int len);

Tests if two string regions are equal. It ignores the case difference if ignoreCase flag is true

21 public boolean startsWith(String prefix, int startIndex)

Tests if this string starts with the specified prefix beginning at specified index. This method does not return any exception but returns false if startIndex is out of bound.

22 Public boolean startsWith(String);

Tests if this string starts with the specified prefix.

23 public boolean endsWith(String suffix)

Tests if this string ends with the specified suffix.

24 public int indexOf(char ch);

Returns the index within this string of the first occurrence of the specified character.

25 public int indexOf(char ch, int fromIndex);

Returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.

26 public int indexOf(String str);

Returns the index within this string of the first occurrence of the specified substring.

27 public int indexOf(String str, int fromIndex);

Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.

28 public int lastIndexOf(int ch);

Returns the index within this string of the last occurrence of the specified character.

29 public int lastIndexOf(int ch, int fromIndex);

Returns the index within this string of the last occurrence of the specified character, searching backward starting the at the specified index.

30 public int lastIndexOf(String str);

Returns the index within this string of the last occurrence of the specified substring.

31 public int lastIndexOf(String str, int fromIndex);

Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

31 public String substring(int);

Gives substring starting from nth character

32 public String substring(int n, int m);

Gives substring starting from nth character up to mth. (not including mth)

33 public String concat(String str);

This will concat the string str at the end of the calling string object.

34 public String replace(char x, char y);

Replace all appearance of x with y

35 public String replaceFirst(String, String);

36 public String replaceAll(String, String);

37 public String toLowerCase();

Converts the string to lowercase

38 public String toUpperCase();

Converts the string to uppercase

39 public String trim();

Removes leading and trailing spaces

40 public char[] toCharArray();

Converts the invoking string to a new character array. This function is provided as a convenience, since it is possible to use getChars() to achieve the same result

41 public static String valueOf(Object);

Create a string object of the parameter p (simple type or object)

String Array:

We can also create and use array that contain strings.

Example 10.1

```
1.     import java.util.Scanner;
2.     class StringArrSort
3.     {
4.         public static void main(String args[])
5.         {
6.             Scanner sc=new Scanner(System.in);
7.             System.out.print("Enter how many names ");
8.             int n=sc.nextInt();
9.             String s[]=new String[n];
10.            //input
11.            for(int i=0;i<n;i++)
12.            {
13.                System.out.println("Enter name " + (i+1));
14.                s[i]=sc.next();
15.            }
16.            //sorting
17.            for(int i=0;i<n-1;i++)
18.                for(int j=i+1;j<n;j++)
19.                    if(s[i].compareTo(s[j]) > 0)
20.                    {
21.                        String t=s[i];
22.                        s[i]=s[j];
23.                        s[j]=t;
24.                    }
25.            //output
26.            for(int i=0;i<n;i++)
27.                System.out.println(s[i]);
28.        }
```

29. }

String Conversion and toString()

When Java converts data into its String representation during concatenation, it does so by calling one of the overloaded versions of the String conversion method `valueOf()` defined by String class. The `valueOf()` is a static method overloaded for all simple types and for type Object.

For the simple types, `valueOf()` returns string that contains the human-readable equivalent of the value with which it is called. For objects, `valueOf()` calls the `toString()` methods on the object.

Every class inherits `toString()`, because it is defined by the Object class. However, the default implementation of the `toString()` is seldom sufficient. It displays name of the class followed by symbol “@” and then object-id (e.g. `Box@ab1342`), which is normally of no use. For most important classes that we create, we will want to override `toString()` and provide our own string representations.

Example 10.2:

```
1.        class A
2.        {
3.            private int x,y;
4.            A()
5.            {
6.                x = y = 0;
7.            }
8.            A(int x1, int y1)
9.            {
10.                x = x1;
11.                y = y1;
12.            }
13.        }
14.        class ToStringTest1
```

```

15.     {
16.         public static void main(String args[])
17.         {
18.             A a1 = new A(5,6);
19.             System.out.println(a1); //toString() of Object class is called
20.             String s1="Object a1 is " + a1; //concat object with string
21.                 //again toString() of Object is called
22.             System.out.println(s1);
23.         }
24.     }

```

Output:

```

A@3e25a5
Object a1 is A@3e25a5

```

Example 10.3:-

```

1.     class A
2.     {
3.         private int x,y;
4.         A()
5.         {
6.             x = y = 0;
7.         }
8.         A(int x1, int y1)
9.         {
10.            x = x1;
11.            y = y1;
12.        }
13.        public String toString()
14.        {
15.            return (x + "," + y);
16.        }
17.    }

```

```

18.     class ToStringTest2
19.     {
20.         public static void main(String args[])
21.     {
22.         A a1 = new A(5,6);
23.         System.out.println(a1); //toString() of A class is called
24.         String s1="Object a1 is " + a1; //concat object with string
25.         //again toString() of A class is called
26.         System.out.println(s1);
27.     }
28.     }

```

Output: 5,6

Object a1 is 5,6

Example 10.4:-

```

1.     import java.util.*;
2.     class FindTest
3.     {
4.         public static void main(String args[])
5.     {
6.         Scanner sc=new Scanner(System.in);
7.         String s1,s2;
8.         System.out.print("Enter a string");
9.         s1=sc.next();
10.        System.out.print("Enter string to search");
11.        s2=sc.next();
12.        int ans = s1.indexOf(s2);
13.        if(ans == -1)
14.            System.out.println("Not Found");
15.        else
16.            System.out.println("found at pos " + ans);
17.    }

```

18. }

StringBuffer:

StringBuffer is a peer class of String that provides much of the functionality of String. As we know, String represents fixed length, immutable character sequence. In contrast to String, StringBuffer represents growable and writeable character sequences. StringBuffer may have characters and substrings inserted in the middle or appended to the end.

StringBuffer will automatically grow to make room for such additions and often has more characters pre-allocated than are actually needed, to allow room for growth. Java uses both class heavily, but many programmer deal only with String and java manipulate StringBuffer behind the scenes by using the overloaded + operator.

Few important point related to StringBuffer class

StringBuffer is an Object like String.

StringBuffer class is also final like String class.

StringBuffer objects are mutable (can be modified) unlike String objects.

Constructors and Methods of StringBuffer class:-

public StringBuffer();

The default constructor reserves room for 16 characters without reallocation.

public StringBuffer(int);

This constructor accept a integer argument that explicitly sets the size of the buffer.

public StringBuffer(String);

This constructor accept a String argument that sets the initial contents of the StringBuffer object and reserves room for 16 more characters without reallocation.

public synchronized int length();

Returns the current length of the StringBuffer.

public synchronized int capacity();

Returns the current capacity. The capacity is the amount of storage available for newly inserted characters, beyond which an allocation will occur.

public synchronized void ensureCapacity(int);

Ensures that the capacity is at least equal to the specified minimum. If we want to pre-allocate room for a certain no. of characters after a StringBuffer has been constructed, we can use ensureCapacity() method to set the size of the buffer. This is useful if we know in advance that we will be appending a large number of small strings to a StringBuffer.

public synchronized void trimToSize();

This will remove extra free space. Now Capacity will be equal to length.

public synchronized void setLength(int newLength);

Sets the length of the character sequence. The newLength must be non-negative. When we increase the size of the buffer, null characters are added to the end of the existing buffer. If we call setLength() with a value less than the current value returned by length(), then the characters stored beyond the new length will be lost.

public synchronized char charAt(int index);

Returns the char value in this sequence at the specified index.

public synchronized void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)

Characters are copied from this sequence into the destination character array dst. The first character to be copied is at index srcBegin; the last character to be copied is at index srcEnd-1. The total number of characters to be copied is srcEnd-srcBegin. The characters are copied into the subarray of dst starting at index dstBegin.

Note: Ensure that the destination or target array is large enough to hold the number of character in the specified substring.

public synchronized void setCharAt(int index, char ch);

The character at the specified index is set to ch. IndexOutOfBoundsException occurs if index is negative or greater than or equal to length().

public synchronized StringBuffer append(Object);

The append method concatenates the string representation of any other type of data to the end of the invoking StringBuffer object.

public synchronized StringBuffer delete(int start, int end);

Removes characters from index start to index end-1

public synchronized StringBuffer deleteCharAt(int index);

Removes the char at the specified position

public synchronized StringBuffer replace(int start, int end, String str);

Replaces the characters from index start to index end-1 with the characters in the specified String.

public synchronized String substring(int);

Returns a new String that contains a subsequence of character currently contained in this character sequence. The substring begins at the specified index and extends to the end of this sequence.

public synchronized String substring(int start, int end);

Returns a new String that contains a subsequence of characters currently contained in this character sequence. The substring begins at the specified start index and extends to the character at index end-1.

public synchronized StringBuffer insert(int, Object);

Insert one String in to another String. It is overloaded to accept values of all the simple types plus Strings and Objects.

public int indexOf(String);

public synchronized int indexOf(String, int);

public int lastIndexOf(String);

public synchronized int lastIndexOf(String, int);

public synchronized StringBuffer reverse();

Reverse the contents of the StringBuffer.

Example 10.5:

1. import java.util.Scanner;
2. class SortStr


```

3.     {
4.         public static void main(String args[])
5.         {
6.             Scanner sc = new Scanner(System.in);
7.             System.out.print("Enter a string");
9.             StringBuffer sb1 = new StringBuffer(sc.next());
10.            //Sorting
11.            for(int i=0;i<sb1.length()-1;i++)
12.                for(int j=i+1;j<sb1.length();j++)
13.                    if(sb1.charAt(i) > sb1.charAt(j))
14.                        {
15.                            char t=sb1.charAt(i);
16.                            sb1.setCharAt(i,sb1.charAt(j));
17.                            sb1.setCharAt(j,t);
18.                        }
19.            System.out.println(sb1);
20.        }
21.    }

```

Output: Enter a string SHIVAM

AHIMSV

Example 10.6: Program to count no. of words in a given sentence.

```

1.     class WordCnt
2.     {
3.         public static void main(String args[])
4.         {
5.             String s1 = "Mohan Das Karam Chand Gandhi";
6.             boolean flag=false;
7.             int cnt=0;
8.             for(int i=0;i<s1.length();i++)
9.                 {
10.                    if(s1.charAt(i) == ' ')
11.                        flag = false;
12.                    else
13.                        {
14.                            if(flag == false)
15.                                {
16.                                    flag = true;

```

```
17.                 cnt++;
18.                 }
19.             }
20.         }
21.         System.out.print("No. of Words=" + cnt);
22.     }
23. }
```

Output: No. of Words=5



CHAPTER

∞ **11** ∞

(Exception Handling)

Introduction-

An Exception is a condition that is caused by a run time error in the program that breaks the normal flow of the program. When the java interpreter encounters an error such as dividing an integer by zero, it creates an Exception object and throws it. (i.e., inform us that an error is occurred).

A Java exception / error is an object of class Exception or one of their sub-classes whenever exception occurs at the run-time. The exception object contains details about the

exception which can be accessed using the public methods provided for this purpose.

In some older languages we have to use `if`, `goto` and return codes for error handling.

An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a runtime error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes, and so on.

This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object-oriented world.

Exception-Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on.

Either way, at some point, the exception is caught and processed. Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**. Briefly, here is how they work. Program statements that you want to monitor for exceptions are contained within a try block. If an exception occurs within the try block, it is thrown. Your code can catch this exception (using `catch`) and handle it in some rational manner. System-generated exceptions are automatically thrown by the Java runtime system. To manually throw an exception, use the keyword `throw`. Any exception that is thrown out of a method must be specified as such by a `throws` clause.

Any code that absolutely must be executed after a try block completes is put in a `finally` block.

This is the general form of an exception-handling block:

This is the general form of an exception handling block:

```
try {  
    // block of code to monitor for errors  
}  
  
catch (ExceptionType1 exOb) {
```

```
// exception handler for Exception Type1
}
//.....
finally {
    // block of code to be executed after try block ends
}
```

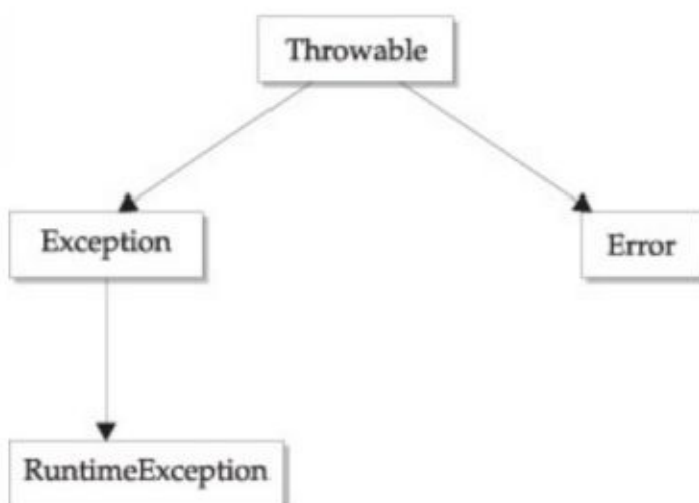
Exception Types

All exception types are subclasses of the built-in class `Throwable`. Thus, `Throwable` is at the top of the exception class hierarchy. Immediately below `Throwable` are two subclasses that partition exceptions into two distinct branches. One branch is headed by `Exception`. This class is used for exceptional conditions that user programs should catch. This is also the class that you will subclass to create your own custom exception types.

There is an important subclass of `Exception`, called **`RuntimeException`**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

The other branch is topped by `Error`, which defines exceptions that are not expected to be caught under normal circumstances by your program. Exceptions of type `Error` are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error. This chapter will not be dealing with exceptions of type `Error`, because these are typically created in response to catastrophic failures that cannot usually be handled by your program.

The top-level exception hierarchy is shown here:



Example 11.1

1. `class ErrorTest1`
2. `{`

```

3.         public static void main(String args[])
4.         {
5.             int a=5,b=2,c=2,d;
6.             d=a/(b-c); //ArithmeticException(Divide by zero) occurred
7.             System.out.println(d);
8.             System.out.println("This will not be printed");
9.         }
10.    }

```

Program will terminate during divide operation and will not run the remaining statements.

Example 11.2

```

1.     class A
2.     {
3.         void display()
4.         {
5.             —
6.         }
7.     }
8.     class ErrorTest2
9.     {
10.        public static void main(String args[])
11.        {
12.            A a1=null;
13.            a1.display(); // NullPointerException occurred
14.        }
15.    }

```

In the above program NullPointerException will occur. The reference variable is not initialized with new. The program gets terminated in this case also.

Example 11.3

```

1.     class ErrorTest3
2.     {
3.         public static void main(String args[])

```

```

4.      {
5.          int a[ ] = {10,20,30};
6.          System.out.println(a[3]); // ArrayIndexOutOfBoundsException
        occurred
7.      }
8.      }

```

In the above case the length of array is 3 and the index is from 0 to 2, so if we print a[3] then this is the 4th element which is not a valid index. This will generate an exception error. In C/C++ it is allowed but not in Java.

NOTE-

JDK 7 adds a new form of the try statement that supports automatic resource management. This new form of try, called **try-with-resources**, is described in file Chapter in the context of managing files because files are some of the most commonly used resources.

Other cases of Exceptions

1. The file we are trying to open may not exist.
2. We want to create a new object but no memory is available.
3. The class file we want to load may be missing or in the wrong format.
4. The String, which we want to convert to a number, is having invalid characters so that it cannot be converted to a number.

If the exception object is not caught and handled properly, the interpreter will display an error message and will terminate the program. If we want to continue with the execution with the remaining code, then we should try to catch the exception object thrown by the error condition.

This task is known as exception handling. The purpose of the exception handling mechanism is to provide a means to detect a report an “exceptional circumstance” so that the appropriate action can be taken. This mechanism suggests incorporation of a separate error handling code that performs the following tasks:

1. Find the problem. (**Exception** occur)
2. Inform that an error has occurred. (**throw** the exception)
3. Receive the error information. (**catch** the exception)
4. The corrective action. (**handle** the exception)

Java **Exception handling** is managed via five **keywords**:

try, catch, throw, throws and finally.

Program statements that we want to monitor for exception are contained within a

try block, and whenever an error occurs then it is thrown to the catch block. Our code can catch this exception (using catch) and handle it in some rational manner. System generated exception are automatically thrown by the java run time system.

To manually throw an exception, use the keyword throw. Any exception that is thrown out of a method must be specified as such by a throws clause. Any code that absolutely must be executed before a method returns is put in a finally block.

The try block must be enclosed between braces even if there is only one statement in it. There can be one or more catch statements and zero or one finally statement. Both catch and finally blocks are optional but either one catch block or one finally block is must.

The code in the try block is executed like any other Java code. If the code inside the try block executes successfully then the control goes to finally block if it is present and then the execution continues from the statement just after the end of finally block (i.e. after the end of try statement) If the finally block is not present then the execution continues from the statement just after the last catch block (i.e. after the end of try statement).

If some run-time error occurs while executing the code in try block then JVM throws an Exception Error. This means an object of type Exception Error or one of its sub-classes is created depending on the type of the run-time error. This is then compared with the Exception/Error types of the catch blocks in top to bottom order.

If a matching catch block is found then the exception / error is handled i.e. program will not terminate. The execution continues with the first statement in the catch block. On completion of the catch block, execution continues with the statement just after the end of try & catch statement.

At the most one catch block is executed irrespective of the number of catch blocks. If exception / error does not match with exception / error type of any of the catch blocks, then we say that it is not handled.

The execution will immediately return from try block. The code in the finally block will be executed even if the exception / error is not handled.

The exception / error will then be handled by outer try block if there is one otherwise it must be handled in the method which called the current method.

The exception / error moves up the hierarchy till it is handled or it is passed to JVM unhandled if not handled even in the main() method which is the first method from which the execution starts.

The JVM then simply terminates the program and displays the exception / error details on the monitor / console.

Here we have represent some Exception which can be occur at the execution time in the program.

Using try and catch-

Although the default exception handler provided by the Java run-time system is

useful for debugging, you will usually want to handle an exception yourself. Doing so provides two benefits. First, it allows you to fix the error.

Second, it prevents the program from automatically terminating. Most users would be confused (to say the least) if your program stopped running and printed a stack trace whenever an error occurred! Fortunately, it is quite easy to prevent this.

To guard against and handle a run-time error, simply enclose the code that you want to monitor inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that you wish to catch.

To illustrate how easily this can be done, the following program includes a try block and a catch clause that processes the `ArithmeticException` generated by the division-by-zero error:

```
class Exc2 {
public static void main(String args[])
{
int d, a;
try {          // monitor a block of code.
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
}
catch (ArithmeticException e)
{
// catch divide-by-zero error
System.out.println("Division by zero.");
}
System.out.println("After catch statement.");
}
}
```

This program generates the following output:

Division by zero.

After catch statement.

Notice that the call to `println()` inside the try block is never executed. Once an exception is thrown, program control transfers out of the try block into the catch block.

Put differently, catch is not "called," so execution never "returns" to the try block from a catch.

Thus, the line “This will not be printed.” is not displayed. Once the catch statement has executed, program control continues with the next line in the program following the entire try/ catch mechanism.

A try and its catch statement form a unit. The scope of the catch clause is restricted to those statements specified by the immediately preceding try statement.

A catch statement cannot catch an exception thrown by another try statement (except in the case of nested try statements, described shortly). The statements that are protected by try must be surrounded by curly braces. (That is, they must be within a block.) You cannot use try on a single statement.

The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened. For example, in the next program each iteration of the for loop obtains two random integers.

Those two integers are divided by each other, and the result is used to divide the value 12345. The final result is put into a. If either division operation causes a divide-by-zero error, it is caught, the value of a is set to zero, and the program continues

```
// Handle an exception and move on.
import java.util.Random;
class HandleError {
public static void main(String args[]) {
int a=0, b=0, c=0;
Random r = new Random();
for(int i=0; i<32000; i++) {
try {
b = r.nextInt();
c = r.nextInt();
a = 12345 / (b/c);
} catch (ArithmeticException e) {
System.out.println(“Division by zero.”);
a = 0; // set a to zero and continue
}
System.out.println(“a: ” + a);
}
} }.
```

Displaying a Description of an Exception

Throwable overrides the toString() method (defined by Object) so that it returns a string

containing a description of the exception. You can display this description in a `println()` statement by simply passing the exception as an argument. For example, the `catch` block in the preceding program can be rewritten like this:

```
catch (ArithmeticException e)
{
    System.out.println("Exception: " + e);
    a = 0; // set a to zero and continue
}
```

When this version is substituted in the program, and the program is run, each divide-by-zero error displays the following message:

Exception: java.lang.ArithmeticException: / by zero

While it is of no particular value in this context, the ability to display a description of an exception is valuable in other circumstances—particularly when you are experimenting with exceptions or when you are debugging.

Exception Type	Cause of the Exception
ArithmeticException	Arithmetic Error such as - Division by 0
ArrayIndexOutOfBoundsException	Out of limit of array indexes
ArrayStoreException	Store the wrong type of data in an array
FileNotFoundException	Access a non-existent file
IOException	Inability to read from a file
NullPointerException	Caused by referencing a null object
NumberFormatException	Conversion between string and number fails
OutOfMemoryException	Not enough memory to allocate a new object
SecurityException	An applets tries to perform an action not allowed by the browser's security setting
StackOverflowException	The System runs out of stack space
StringIndexOutOfBoundsException	A program attempt to access a nonexistent character position in a string

Java's Built-in Exceptions

Inside the standard package `java.lang`, Java defines several exception classes. A few have been used by the preceding examples. The most general of these exceptions

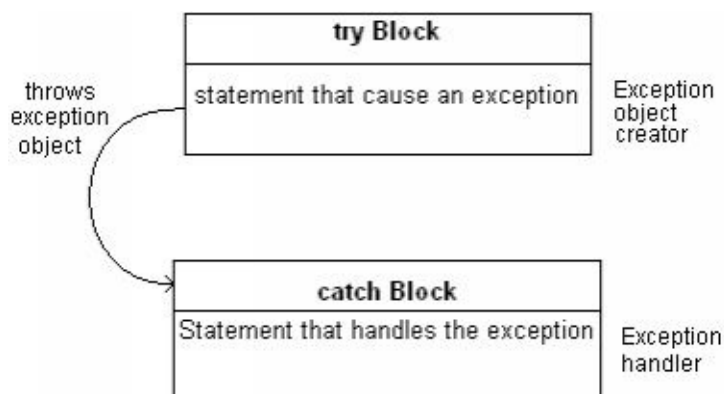
are subclasses of the standard type RuntimeException.

As previously explained, these exceptions need not be included in any method's throws list. In the language of Java, these are called unchecked exceptions because the compiler does not check to see if a method handles or throws these exceptions. The unchecked exceptions defined in java.lang are listed in Table -1. Table -2 lists those exceptions defined by java.lang that must be included in a method's throws list if that method can generate one of these exceptions and does not handle it itself. These are called checked exceptions. Java defines several other types of exceptions that relate to its various class libraries.

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBoundsException	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table Java's Unchecked RuntimeException
Subclasses Defined in java.lang

General form of an Exception handling



(Exception handling mechanism)

Syntax:

```
-----  
-----  
try  
{  
    statement;           //generates an exception  
}  
catch (Exception-type e)  
{  
    statement;           //process the exception  
}  
-----  
-----
```

Multiple catch Clauses

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.

When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try / catch block. The following example traps two different exception types:

```
// Demonstrate multiple catch statements.  
class MultipleCatches {  
    public static void main(String args[]) {  
        try {  
            int a = args.length;  
            System.out.println("a = " + a);  
            int b = 42 / a;  
            int c[ ] = { 1 };  
            c[42] = 99;  
        } catch(ArithmeticException e) {
```

```

System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
System.out.println("Array index oob: " + e);
}
System.out.println("After try/catch blocks.");
}
}

```

This program will cause a division-by-zero exception if it is started with no command-line arguments, since a will equal zero. It will survive the division if you provide a command line argument, setting a to something larger than zero. But it will cause an `ArrayIndexOutOfBoundsException`, since the int array c has a length of 1, yet the program attempts to assign a value to `c[42]`.

Here is the output generated by running it both ways:

```
C:\>java MultipleCatches
```

```
a = 0
```

```
Divide by 0: java.lang.ArithmeticException: / by zero
```

```
After try/catch blocks.
```

```
C:\>java MultipleCatches TestArg
```

```
a = 1
```

```
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42 After try/catch blocks.
```

When you use multiple catch statements, it is important to remember that exception subclasses must come before any of their superclasses. This is because a catch statement that uses a superclass will catch exceptions of that type plus any of its subclasses.

Thus, a subclass would never be reached if it came after its superclass. Further, in Java, unreachable code is an error. For example, consider the following program:

```

/* This program contains an error.
A subclass must come before its superclass in
a series of catch statements. If not,
unreachable code will be created and a
compile-time error will result.
*/

```

```

class SuperSubCatch {
public static void main(String args[]) {
try {
int a = 0;
int b = 42 / a;
} catch(Exception e) {
System.out.println("Generic Exception catch.");
}
/* This catch is never reached because
ArithmeticException is a subclass of Exception. */
catch(ArithmeticException e) { // ERROR – unreachable
System.out.println("This is never reached.");
}
}
}
}

```

If you try to compile this program, you will receive an error message stating that the second catch statement is unreachable because the exception has already been caught. Since `ArithmeticException` is a subclass of `Exception`, the first catch statement will handle all `Exception`-based errors, including `ArithmeticException`.

This means that the second catch statement will never execute. To fix the problem, reverse the order of the catch statements.

Nested try Statements

The `try` statement can be nested. That is, a `try` statement can be inside the block of another `try`. Each time a `try` statement is entered, the context of that exception is pushed on the stack.

If an inner `try` statement does not have a catch handler for a particular exception, the stack is unwound and the next `try` statement's catch handlers are inspected for a match.

This continues until one of the catch statements succeeds, or until all of the nested `try` statements are exhausted. If no catch statement matches, then the Java runtime system will handle the exception.

Here is an example that uses nested `try` statements:

Multiple catch statement

When there are more than one catch statement then it is said to be a multiple catch statement.

Example multiple catch statements

```
try
{
    statement;           //generate an exception
}
catch(Exception-type-1 e)
{
    statement 1;        //process exception type1
}
catch(Exception-type-2 e)
{
    statement 2;        //process exception type2
}
--
--
catch(Exception-type-n e)
{
    statement n;        //process exception type n
}
```

Example 11.4

```
1.      class ErrorTest4
2.      {
3.          public static void main(String args[])
4.          {
5.              int a=10,b=2,c=2,d;
6.              System.out.println("Before Exception ");
7.              d=a/(b-c);
8.              System.out.println("After Exception ");
9.          }
10.     }
```

Output :

```
Before Exception
Exception in thread "main"java.lang.
ArithmeticException:/ by zero
at Arith_Excep.main(Arith_Excep.java:8)
```

Here we are not catching the exception so the program will terminate, after displaying the message “Before Exception”. Now we have solved this problem using exception handling.

Example 11.5

```
1.      class ErrorTest5
2.      {
3.          public static void main(String[] args)
4.          {
5.              int a=10,b=2,c=2,d;
6.              try
7.              {
8.                  System.out.println(“Before Exception”);
9.                  d=a/(b-c);
10.                 System.out.println(“This Will Not Print”);
11.             }
12.             catch(ArithmeticException e)
13.             {
14.                 System.out.println(“After Exception”);
15.                 System.out.println(“error : Division by zero”);
16.             }
17.         }
18.     }
```

Output:

```
Before Exception
After Exception
error : Division by zero
```

Advantages of Exception Handling-

- We can easily say where the exception / error will be handled. Exception / Errors propagate up the call stack at runtime- first up the enclosing try blocks and then back to the calling method-until an exception handler catch them.

- The location of the exception / error is known exactly as entire stack trace is available to the user. This is very helpful in debugging.
- Programmer gets a chance to recover from the error / abnormal condition.
- Error handling code is separated from the normal program flow to increase the readability and maintainability
- With java there is no need to test if an exception / error condition happens. Adding more error / exception handler simply requires adding more catch clauses, but the original program flow need not be touched.

Hierarchy of Exception Class-

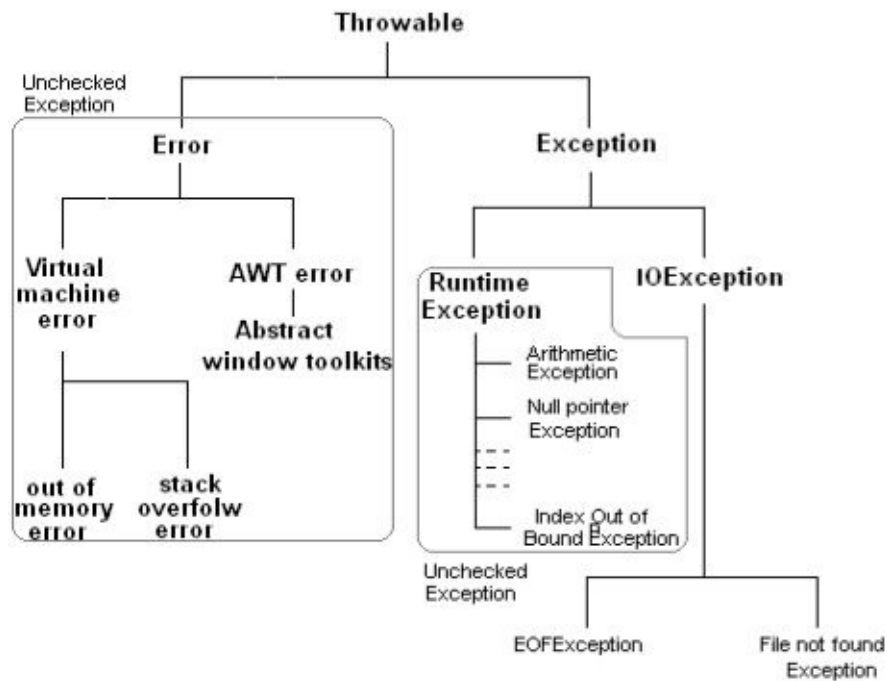
Root class of all Exception / Error class is the **throwable** class, which is an immediate subclass of the object class. Exceptions/Errors are also objects in Java. Sub classes of Exception have the suffix Exception.

There are two immediate subclass of the class throwable.

- **Error:-**The sub classes of Error class are basically used for signaling abnormal system conditions like:
 1. **OutOfMemoryError** signals that the Java VM has run out of memory and that the garbage collector is unable to claim any free memory.
 2. **StackOverflowError** signals a stack overflow in the interpreter.

The errors are, in general, unrecoverable and should not be handled.
- **Exception:-**The sub classes of Exception class are, in general recoverable. For example, **EOFException** signals that a file we have opened has no more data for reading. **FileNotFoundException** signals that a file we want to open does not exist in the file system.

Example of the Error hierarchy :



Exception categorized in two types

- **Checked Exception.**
- **Unchecked Exception.**

Checked Exception

If it is must to catch an exception then it is called a checked exception. The program will not compile if there is a possibility of occurring checked exception in a try block and it is not handled.

The compiler ensured that if a method can throw a checked exception, directly or indirectly, then the method must explicitly deal with it. The method must either catch the exception and take the appropriate action, or pass the exception on to its caller.

Note: Exception and all of its sub-class (Excluding run time-exception and its subclass) are checked and must be caught.

Unchecked Exception

If it is not must to catch an exception then its called an unchecked exception. The program will compile even if the unchecked exception is not handled. If any unchecked exception occurs in a program which is not handled then the program execution will terminate at that point. But if exception is handled then program will not terminate because of exception.

Note:-Error and all of its subclass are unchecked exception and need not be caught. Run-time Exception and all of its subclass are unchecked exception and need not be caught. All user defined Exception are checked.

Methods available in Exception class-

String getMessage()	To obtain the error message associated with exception or error.
---------------------	---

void printStackTrace()	To print a stack trace showing where the exception/error occurs.
String toString()	To show the exception/error name along with the message returned by getMessage().

Example 11.6

```

1.      class ErrorTest6
2.      {
3.          public static void main(String[] arr)
4.          {
5.              try
6.              {
7.                  int a=10,b=2,c=2,d;
8.                  System.out.println("Before Exception");
9.                  d=a/(b-c);
10.                 System.out.println("This Will Not Print");
11.             }
12.             catch(ArithmeticException e)
13.             {
14.                 System.out.println("After Exception");
15.                 System.out.println("error : Division by zero");
16.                 System.out.println(e.getMessage());
17.                 System.out.println(e); //toString() is called.
18.                 e.printStackTrace();
19.             }
20.         }
21.     }

```

Output:

```

Before Exception
After Exception
error : Division by zero
/ by Zero

```

```
java.lang.ArithmeticException: / by zero
java.lang.ArithmeticException: / by zero
at ErrorTest6.main(ErrorTest6.java:10)
```

Example 11.7 Multiple catches

```
1.      class ErrorTest7
2.      {
3.          public static void main(String args[ ])
4.          {
5.              try
6.              {
7.                  int a=args.length;
8.                  System.out.println("Before Exception");
9.                  int b=58/a;
10.                 int c[ ] = {1};
11.                 c[47]=100;
12.                 System.out.println("After Exception");
13.             }
14.             catch(ArithmeticException e)
15.             {
16.                 System.out.println("Divide by zero :"+e);
17.             }
18.             catch(ArrayIndexOutOfBoundsException e)
19.             {
20.                 System.out.println("Array Index oob :"+e);
21.             }
22.             System.out.println("After try/catch block");
23.         }
24.     }
```

Output 1: When we run the above program as "java ErrorTest7"

Before Exception

Divide by zero : java.lang.ArithmeticException: / by zero

After try/catch block

Output 2: When we run the above program as "java ErrorTest7 45"

Before Exception

After try/catch block

Nested try catch

If we use try and catch statements in another try block then it is said to be a nested try catch.

Syntax

```

-----
try                                //outer try
{
-----
try                                //inner try
{
statement ;
}
catch1(-----)                    //inner catch
{
statement ;
}
catch2(-----)                    //inner catch
{
statement ;
}
}
catch(-----)                     //outer catch
{
statement ;
}
-----

```

Example 11.8 Nested try and catch-

```

1.      class Nested
2.      {
3.          public static void main(String args[])
4.          {
5.              try
6.              {

```

```

7.         int a=args.length;
8.         int b=58/a;
9.         System.out.println("a= "+a);
10.        try
11.            {
12.                if(a==1)
13.                    a=a/(a-a);
14.                if(a==2)
15.                    {
16.                        int c[]={1};
17.                        c[42]=100;
18.                    }
19.            }
20.        catch(ArrayIndexOutOfBoundsException e)
21.            {
22.                System.out.println("ArrayIndex oob "+e);
23.            }
24.        }
25.        catch(ArithmeticException e)
26.            {
27.                System.out.println(e.getMessage());
28.            }
29.        }
30.    }

```

Output I:

When we run the above program as "java ErrorTest8" / by zero

Output II:

When we run the above program as "java ErrorTest8 10"

a=1 / by zero

Output III:

When we run the above program as "java ErrorTest8 10 20" a=2

ArrayIndex oob java.lang.ArrayIndexOutOfBoundsException: 42

finally Statement

Java support another statement known as finally statement that can be used to handle an exception that is not caught by any of the previous catch statement. **finally** block can be used to handle any exception generated within a try block. It will executed in all the cases whether error occurred or not. It may be added immediately after the try block or after the last catch block shown as follows:

Syntax:

```
try
{
    -----
}
catch(... )
{
    -----
}
catch(... )
{
    -----
}
finally
{
    -----
}
```

Example 11.9 finally statement

```
1.      class ErrorTest9
2.      {
3.          public static void main(String args[])
4.              {
5.                  int a[ ] = {5,10};
```

```

6.         int b=5;
7.         try
8.         {
9.             int x=a[2]/b-a[1];
10.        }
11.        catch(ArithmeticException e)
12.        {
13.            System.out.println("Division By Zero");
14.        }
15.        catch(ArrayIndexOutOfBoundsException e)
16.        {
17.            System.out.println("Array Index error");
18.        }
19.        catch(ArrayStoreException e)
20.        {
21.            System.out.println("Wrong Data Type");
22.        }
23.        finally          //always execute
24.        {
25.            System.out.println("Always Execute");
26.        }
27.        int y=a[1]/a[0];
28.        System.out.println("y = "+y);
29.    }
30. }

```

Output: Array Index error

Always Execute

y = 2

throw

So far, we are only been catching exception that are thrown by the java run-time system. However its possible for our program to throw an exception explicitly, using the throw statement, the general form of throw is shown here:

throw *Throwableinstance*;

Here, Throwableinstance must be an object of type **Throwable** or a subclass of **Throwable**. Simple type such as **int** or **char**, as well as non-Throwable classes such as **String** and **Object**, cannot be used as exception. There are two ways we can obtain a **Throwable** object: using a parameter into a **catch** clause, or creating one with the **new** operator

The flow of execution stops immediately after the **throw** statement; any subsequent statement are not executed. The nearest enclosing try block is inspected to see if it has a **catch** statement that matches the type of the exception. If it does find a match, control is transferred to that statement. If not then the next enclosing try statement is inspected, and so on. If no matching **catch** is found, then the default exception handler halts the program prints the stack trace.

Here we have taken a program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

Example 11.10

```
1.      class ErrorTest10
2.      {
3.          int age;
4.          void setAge(int a)
5.          {
6.              if(a>0)
7.                  age = a;
8.              else
9.                  {
10.             NullPointerException e = new NullPointerException("Invalid Age");
11.             throw e;
12.             //throw new NullPointerException("Invalid Age");
13.             }
14.         }
15.         public static void main(String args[])
16.         {
17.             ErrorTest10 a1 = new ErrorTest10();
18.             try
19.             {
```

```

20.         a1.setAge(20);
21.         a1.setAge(-10);
22.         }
23.         catch(NullPointerException e)
24.         {
25.             System.out.println(e.getMessage());
26.         }
27.     }
28. }

```

Output:

Invalid Age

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that caller of the method can guard themselves against that exception. We do this by including a throws clause in the method's declaration. A throws clause lists the type of exception that a method might throw.

This is necessary for all checked exceptions. All exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result. This is a general form of a method declaration that includes a throws clause:

Syntax:

```

type method-name(parameter list) throws exception-list
{
    //body of method
}

```

Here, exception list is a comma-separated list of the exception that a method can throw

Example 11.11

```

0.     import java.io.*;
1.     class ErrorTest11
2.     {
3.         int age;

```

```

4.         void setAge(int a) throws IOException
5.         {
6.             if(a>0)
7.                 age = a;
8.             else
9.                 {
10.                IOException e = new IOException("Invalid Age");
11.                throw e;
12.            }
13.        }
14.        public static void main(String args[])
15.        {
16.            ErrorTest11 a1 = new ErrorTest11();
17.            try
18.            {
19.                a1.setAge(20);
20.                a1.setAge(-10);
21.            }
22.            catch(IOException e)
23.            {
24.                System.out.println(e.getMessage());
25.            }
26.        }
27.    }

```

Output:

Invalid Age

Creating user-defined Exception/Error sub-classes:

Creating Your Own Exception Subclasses-

This is quite easy to do, just define a sub class of Exception/Error class. Our sub classes do not need to actually implement anything. The Exception/Error class does not define any methods of its own. It does, of course, inherit methods provided by Throwable class.

Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications. This is quite easy to do: just define a subclass of `Exception` (which is, of course, a subclass of `Throwable`).

Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions.

The `Exception` class does not define any methods of its own. It does, of course, inherit those methods provided by `Throwable`. Thus, all exceptions, including those that you create, have the methods defined by `Throwable` available to them.

`Exception` defines four constructors. Two support chained exceptions, described in the next section. The other two are shown here:

`Exception()`

`Exception(String msg)`

The first form creates an exception that has no description. The second form lets you specify a description of the exception.

Although specifying a description when an exception is created is often useful, sometimes it is better to override `toString()`. Here's why: The version of `toString()` defined by `Throwable` (and inherited by `Exception`)

first displays the name of the exception followed by a colon, which is then followed by your description. By overriding `toString()`, you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

Example 11.12

```
1.      class AgeException extends Exception
2.      {
3.          AgeException(String msg)
4.          {
5.              super(msg);
6.          }
7.      }
8.      class ErrorTest12
9.      {
10.         int age;
11.         void setAge(int a) throws InvalidAgeException
12.         {
13.             if(a>0)
```

```

14.             age = a;
15.         else
16.         {
17.             AgeException e = new AgeException("Invalid Age");
18.             throw e;
19.         }
20.     }
21.     public static void main(String args[])
22.     {
23.         ErrorTest12 a1 = new ErrorTest12();
24.         try
25.         {
26.             a1.setAge(20);
27.             a1.setAge(-10);
28.         }
29.         catch(AgeException e)
30.         {
31.             System.out.println(e.getMessage());
32.         }
33.     }
34. }

```

Output:

Invalid Age

The following example declares a new subclass of Exception and then uses that subclass to signal an error condition in a method. It overrides the toString() method, allowing a carefully tailored description of the exception to be displayed.

```

// This program creates a custom exception type.
class MyException extends Exception {
    private int detail;

    MyException(int a) {
        detail = a;
    }

    public String toString() {
        return "MyException[" + detail + "]";
    }
}

class ExceptionDemo {
    static void compute(int a) throws MyException {
        System.out.println("Called compute(" + a + ")");
        if(a > 10)
            throw new MyException(a);
        System.out.println("Normal exit");
    }

    public static void main(String args[]) {
        try {
            compute(1);
            compute(20);
        } catch (MyException e) {
            System.out.println("Caught " + e);
        }
    }
}

```

This example defines a subclass of Exception called MyException. This subclass is quite simple: It has only a constructor plus an overridden toString() method that displays the value of the exception.

The ExceptionDemo class defines a method named compute() that throws a MyException object. The exception is thrown when compute()'s integer parameter is greater than 10. The main() method sets up an exception handler for MyException, then calls compute() with a legal value (less than 10) and an illegal one to show both paths through the code.

Here is the result:

Called compute(1)

Normal exit

Called compute(20)

Caught MyException[20]

Chained Exceptions:

The chained exception feature allows us to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an ArithmeticException because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly.

Although the method must certainly throw an `ArithmeticException`, since that is the error that occurred We might also want to let the calling code know that the underlying cause was as I/O error.

To allow chained exceptions, Java 2, version 1.4 added two constructors and two methods to `Throwable` class. The constructors are shown here:

`Throwable(Throwable causeExc)`

`Throwable(String msg, Throwable causeExc)`

In the first form, `causeExc` is the exception occurred. The second form allows us to specify a description at the same time that we specify a cause exception. These two constructors have also been added to the `Error`, `Exception`, and `RuntimeException` classes.

The chained exception methods added to `Throwable` class are `getCause()` and `initCause()`

`Throwable getCause()`

`Throwable initCause(Throwable causeExc)`

The `getCause()` method returns the exception that underlies the current exception. If there is no underlying exception, `null` is returned. The `initCause()` method associates `causeExc` with the invoking exception and returns a reference to the exception. Thus, we can associate a cause with an exception after the exception has created. However, the cause exception can be set only once.

Thus we can call `initCause()` only once for each exception object. Furthermore, if the cause exception was set by a constructor, then we can not set it again using `initCause()` method. In general, `initCause()` is used to set a cause for legacy exception classes which do not support the two additional constructors described earlier. Most of Java's built-in exceptions do not define the additional constructors. Thus, we will use `initCause()` if we need to add an exception chain to these exceptions.

Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

Example11.13

```
1.      Class ErrorTest13
2.      {
3.          int age;
4.          void setAge(int a)
5.          {
6.              if(a>0)
7.                  age = a;
8.          else
```

```
9.         {
10.         NullPointerException e = new NullPointerException("Admission Failed");
11.         e.initCause(new ArithmeticException("Age is invalid"));
12.         throw e;
13.         }
14.     }
15.     public static void main(String args[ ])
16.     {
17.         ErrorTest13 a1 = new ErrorTest13();
18.         try
19.         {
20.             a1.setAge(20);
21.             a1.setAge(-10);
22.         }
23.         catch(NullPointerException e)
24.         {
25.             System.out.println(e.getMessage());
26.             System.out.println(e.getCause());
27.         }
28.     }
29. }
```

Output:

Admission Failed

Age is Invalid


```

// Demonstrate exception chaining.
class ChainExcDemo {
    static void demoproc() {

        // create an exception
        NullPointerException e =
            new NullPointerException("top layer");

        // add a cause
        e.initCause(new ArithmeticException("cause"));

        throw e;
    }
    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            // display top level exception
            System.out.println("Caught: " + e);

            // display cause exception
            System.out.println("Original cause: " +
                e.getCause());
        }
    }
}

```

The output from the program is shown here:

```

Caught: java.lang.NullPointerException: top layer
Original cause: java.lang.ArithmeticException: cause

```

In this example, the top-level exception is `NullPointerException`. To it is added a cause exception, `ArithmeticException`. When the exception is thrown out of `demoproc()`, it is caught by `main()`. There, the top-level exception is displayed, followed by the underlying exception, which is obtained by calling `getCause()`.

Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design. Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

Three New JDK 7 Exception Features (UPDATED)

JDK 7 adds three interesting and useful features to the exception system. The first automates the process of releasing a resource, such as a file, when it is no longer needed.

It is based on an expanded form of the try statement called **try-with-resources**, and is described in upcoming Chapter when files are introduced.

The second new feature is called **multi-catch**, and the third is sometimes referred to as final rethrow or more precise rethrow. These two features are described here.

The multi-catch feature allows two or more exceptions to be caught by the same catch clause. It is not uncommon for two or more exception handlers to use the same code sequence even though they respond to different exceptions.

Instead of having to catch each exception type individually, now you can use a single catch clause to handle all of the exceptions without code duplication.

To use a multi-catch, separate each exception type in the catch clause with the OR operator. Each multi-catch parameter is implicitly final. (You can explicitly specify final, if desired, but it is not necessary.)

Because each multi-catch parameter is implicitly final, it can't be assigned a new value.

Here is a catch statement that uses the **multi-catch feature** to catch both **ArithmeticException** and **ArrayIndexOutOfBoundsException**:

```
catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
```

The following program shows the multi-catch feature in action:

```
// Demonstrate JDK 7's multi-catch feature.
```

```
class MultiCatch {
```

```
public static void main(String args[]) {
```

```
int a=10, b=0;
```

```
int vals[] = { 1, 2, 3 };
```

```
try {
```

```
int result = a / b; // generate an ArithmeticException
```

```
// vals[10] = 19; // generate an ArrayIndexOutOfBoundsException
```

```
// This catch clause catches both exceptions.
```

```
} catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
```

```
System.out.println("Exception caught: " + e);
```

```
}
```

```
System.out.println("After multi-catch.");
```

```
}}
```

The program will generate an `ArithmeticException` when the division by zero is attempted. If you comment out the division statement and remove the comment symbol from the next line, an `ArrayIndexOutOfBoundsException` is generated. Both exceptions are caught by the single catch statement.

The more precise `rethrow` feature restricts the type of exceptions that can be rethrown to only those checked exceptions that the associated try block throws, that are not handled by a preceding catch clause, and that are a subtype or supertype of

the parameter.

Although this capability might not be needed often, it is now available for use. For the more precise rethrow feature to be in force, the catch parameter must be either effectively final, which means that it must not be assigned a new value inside the catch block, or explicitly declared final.

Using Exceptions

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It is important to think of try, throw, and catch as clean ways to handle errors and unusual boundary conditions in your program's logic. Unlike some other languages in which error return codes are used to indicate failure, Java uses exceptions. Thus, when a method can fail, have it throw an exception.

This is a cleaner way to handle failure modes. One last point: Java's exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.

Theory Question:

1. What is an exception?
2. How do we define a try block?
3. How do we define a catch block?
4. List some of the most common types of exception that might occur in java. Give Example.
5. Is it essential to catch all types of exception?
6. How many catch blocks can we use with one try block.
7. Create a try block that is likely to generate three types of exception and then incorporate necessary catch block to catch and handle them appropriately.
8. What is finally block? When and how is it used? Give a suitable example.
9. Explain how Exception handling mechanism can be used for debugging a program.





CHAPTER

∞ 12 ∞

(Multi-Threaded Programming)

Introduction-

A thread is a single flow of control within a program. Thread is very much similar to a process. In fact thread is also called a light-weight process.

Thread v/s Process:

Normally different processes occupy different memory space. When the CPU shifts from one process to another process, the state of the currently running process is saved and the state of another process is restored. No useful work is being done during state switch. The context switch time should be as less as possible to maximize the CPU utilization.

Threads are also like independent processes but they share the same memory and state. The separate threads also have separate state but the state is very small as compared to process state. The state may contain just program counter and stack pointer. So switching from one thread to another thread takes very little time. Thus the context switch time for threads is very small as compared to the process. Hence CPU utilization is high in case of multiple threads as compared to the utilization in case of multiple processes.

Multi-Threaded program:

A multi-threaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multi-threading is a specialized form of multi-tasking.

For example, a program may have three threads:

- One handling the printing.
- One handling the editing.
- One downloading a file from the Internet.

All these threads might be running concurrently thus maximizing the CPU utilization.

Process-based Multi-tasking:

Most of the operating systems allow us to run two or more programs at the same time. It is referred to as process-based multi-tasking. For example, we can run a Java program and at the same time we may be editing a word document. The process-based multi-tasking ensures that there will be some program to be executed most of the time so it increases the CPU utilization.

In process-based multi-tasking, a program is the smallest unit of code that can be dispatched by the scheduler.

Thread-based multi-tasking:

Thread is the smallest unit of execution in a thread-based multi-tasking environment. A process can be divided into a number of threads executing concurrently. This allows us to handle more than one task concurrently in a single program.

For example, we can edit a word document and at the same time print another document. Thread-based multi-tasking improves CPU utilization just like process-based multi-tasking. But at the same time it effectively speeds up the execution of a single program by executing its different parts concurrently in separate threads.

Thus process based multi-tasking deals with the “big picture”, and thread-based multi-tasking handles the details.

The Java Thread Model:

The Java run-time system depends on threads for many things, and all the class libraries are designed with multi-threading in mind. In fact, Java uses threads to enable the entire environment to be asynchronous. This helps reduce inefficiency by preventing the waste of CPU cycles.

Single-threaded systems use an approach called an event loop with polling. In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next. In a single threaded environment, when a thread blocks (that is, suspends execution) because it is waiting for some resource, the entire program stops running.

The benefit of Java’s multi-threading is that the main loop/polling mechanism is

eliminated. When a thread blocks in Java program, only the single thread that is blocked pauses, all other threads continue to run.

Thread Life Cycle (Thread States):

Thread exists in several states. A thread can be running. It can be ready to run as soon as it gets CPU time. A running thread can be suspended, which temporarily suspends its activity. A suspended thread can then be resumed, allowing it to pick up where it left off. A thread can be blocked when waiting for a resource. At any time, a thread can be terminated, which halts its execution immediately. Once terminated a thread cannot be resumed.

1. Newborn State:

When we create a thread object, the thread is born and is said to be in newborn state. The thread is not yet scheduled for running.

At this state, we can do only one of the following things with it:

- 1. Schedule it for running using start() method.**
- 2. Kill it using stop() method.**

If scheduled, it moves to runnable state. If we attempt to use any other method at this state, an exception will be thrown.

2. Runnable State:

The runnable state means that the thread is ready for execution and is waiting for the availability of the processor. That is, the thread has joined the queue of threads that are waiting for execution. If all threads are of equal priority, then they are given time slots for execution in round robin fashion.

The thread that relinquishes control joins the queue at the end and again waits for its run. However, if we want a thread to relinquish control to another thread of equal priority before its turn comes, it can do so by invoking the yield() method.

3. Running State:

Running means that the processor has given its time to the thread for its execution. A running thread may relinquish its control in one of the following situations:

- Its time-slice is over.
- It is pre-empted by a higher priority thread.
- It yields i.e. voluntarily relinquishes control.
- It has completed its execution.
- It is stopped by some other thread.
- It has been suspended using suspend() method. A suspended thread can be revived by using the resume() method. This approach is useful when we want to

suspend a thread for some time due to certain reason, but do not want to kill it.

- It has been told to wait until some event occurs. This is done using the wait() method. The thread can be scheduled to run again using the notify() method.
- It is performing some I/O operation.

4. Blocked State:

A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state. This happens when the thread is suspended, sleeping or waiting in order to satisfy certain requirements. A blocked thread is considered “not runnable” but not dead and therefore fully qualified to run again.

5. Dead State:

Every thread has a life cycle. A running thread ends its life when it has completed executing its run() method. It has a natural death. However, we kill it by sending the stop message to it at any state thus causing a premature death. A thread can be killed as soon as it is born, or while it is running, or even when it is in “not runnable” (blocked condition).

Thread Priorities:

Java assigns to each thread a priority that determines how that thread should be treated with respect to the others. Thread priorities are integers that specify the relative priority of one thread to another.

As an absolute value, a priority is meaningless; a higher priority thread does not run any faster than a lower-priority thread if it is the only thread running. Instead, a thread’s priority is used to decide when to switch from one running thread to next. This is called a context switch. The rules that determine when a context switch takes place are simple:

- A thread can voluntarily (on its own) relinquish control. This is done by explicitly yielding, sleeping or blocking or pending I/O. In this scenario, all other threads are examined, and normally the highest-priority thread that is ready to run is given the CPU.
- A thread can be pre-empted by a higher-priority thread. In this case, a lower priority thread that does not yield the processor is simply pre-empted no matter what it is doing, by a higher priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called **preemptive multitasking**.
- Some OS support non-preemptive priority based scheduling. In such case a high priority thread gets chance only when low priority thread completes.
- In cases where two threads with the same priority are competing for CPU cycles, the situation is a bit complicated. For OS such as windows98, threads of equal priority must voluntarily (on their own) yield (give up) control to their peers. If

they do not, the other threads will not run.

Note:- Problems can arise from the differences in the way that O.S.'s context-switch threads of equal priority.

Synchronization:

Because multi-threading introduces as asynchronous behavior to our programs, there must be a way for us to enforce synchronization when we need it. For example, if we want two threads to communicate and share a complicated data structure, such as a linked list, we need some way to ensure that they do not conflict with each other.

That is, we must prevent one thread from writing data while another thread is in the middle of reading it. Java uses monitor for inter-thread synchronization. We can think of a monitor as a very small box that can hold only one thread. Once a thread enters a monitor can be used to protect a shared asset from being manipulated by more than one thread at a time.

Most multi-threaded systems expose as objects that our program must explicitly acquire and lock. Java provides a cleaner solution.

There is no class "monitor", instead, each object has its own implicit monitor that is automatically entered when one of the object's synchronized method is called. Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object. This enables us to write very clear and concise multi-threaded code, because synchronization support is built into the language.

Messaging: When programming with most other languages, we must depend on the O.S. to establish communication between threads.

This, of course, adds overhead. By contrast, Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have. Java's messaging system allows a thread to enter synchronized method on an object, and then wait there until some other thread explicitly notifies to come out.

The Thread class and the Runnable interface:

Java's multi-threading system is built upon the Thread class, its methods, and its companion interface, Runnable. This class belongs to package java.lang and hence there is no need of explicitly importing it.

Thread encapsulates a thread of execution, since we cannot directly refer to the internal state of a running thread, we will deal with it through its proxy, the Thread instance that spawned it. To create a new thread our program will either extend Thread class or implement the Runnable interface. The Thread class defines several methods that help manage threads:

String getName()

Returns this thread's name

int getPriority()

Returns this thread's priority

boolean isAlive()

Tests if this thread is still running

void join()

Waits for this thread to die (terminate)

void run()

If this thread was constructed using a separate Runnable object, then that Runnable object's run method is called; otherwise, this method does nothing and returns, if thread class is extended and run() method is overridden in sub-class then the overridden run() method is called.

void setName(String name)

Changes the name of this thread to be equal to the argument name

static void sleep(long millis) throws InterruptedException

Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.(1 sec = 1000ms)

static void sleep(long millis, int nanos) throws InterruptedException

Causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds.

void start()

Causes this thread to begin execution, the Java Virtual Machine calls the run method of this thread.

static void yield()

Causes the currently executing thread object to temporarily pause and allow other threads to execute.

static Thread currentThread()

Returns a reference to the currently executing thread object.

The main thread:

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of our program, because it is the one that is executed when our program begins. The main thread is the thread from which other "child" threads are

created.

Although the main thread is created automatically when our program is started, it can be controlled through a Thread object. To do so, we must obtain a reference to it by calling the method `currentThread()`, which is public static member of Thread class.

This method returns a reference to the thread in which it is called. Once we have reference to the main method, we can control it just like any other thread.

Example 12.1

The following example demonstrates how we can acquire reference of main thread and then access its properties using methods of Thread class.

```
1.      class CurrentThreadTest
2.      {
3.          public static void main (String args[])
4.              {
5.              Thread t = Thread.currentThread();
6.              System.out.println("Current thread:" + t);
7.              System.out.println("Name:" + t.getName());
8.              System.out.println("Priority:" + t.getPriority());
9.              t.setName("MyThread");
10.             t.setPriority(Thread.MAX_PRIORITY);
11.             System.out.println("After name and priority change :" + t);
12.             System.out.println("Name:" + t.getName());
13.             System.out.println("Priority:" + t.getPriority());
14.             for (int n=1; n<=5; n++)
15.                 {
16.                 System.out.println(n);
17.                 try
18.                 {
19.                     Thread.sleep(1000);
20.                 }
21.                 catch (InterruptedException e)
22.                 {
23.                     System.out.println("Main thread interrupted");
24.                 }
```

```
25.         }
26.     }
27. }
```

Output:

```
Current thread:Thread[main,5,main]
Name:main
Priority:5
After name and priority change :Thread[MyThread,10,main]
Name:MyThread
Priority:10
1
2
3
4
5
```

Note 1: The sleep() method in Thread might throw an InterruptedException, which is a checked exception. This would happen if some other thread wanted to interrupt this sleeping one.

Note 2: Notice the output produced when t (thread reference) is used as an argument to println(). This will display in order: the name of the thread, its priority, and the name of its group. Its priority is 5, which is the default value, and main is also the name of the group of thread to which this thread belongs. A thread group is a data structure that controls the state of a collection of threads as a whole.

Creating a Thread:

In the most general sense, we create a thread by instantiating an object of type Thread. Java identifies two ways in which this can be accomplished:

- 1. We can implement the Runnable interface**
- 2. We can extend the Thread class, itself.**

Implementing Runnable:

The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called run().

```
public void run( )
```

Inside run() method, we will define the code that constitutes the new thread. The

run() can call other methods, use other classes and declare variables, just like the main thread.

The only difference is that run() establishes the entry point for another, concurrent thread of execution within our program. This thread will end when run() returns.

After we create a class that implements Runnable, we will instantiate an object of type Thread from within that class using one of the following constructors:

Thread(Runnable threadObj)

Thread(Runnable threadObj, String threadName)

Here threadObj is the object whose run method is called and threadName is the name of the new thread. After the new thread is created, it will not start running until we call start() method.

The start() method puts the thread in the ready queue (runnable state). Whenever the thread gets scheduled its execution will start from the run() method.

Example12.2

```
1.      class A implements Runnable
2.      {
3.          public void run()
4.          {
5.              for(int i=1;i<=5;i++)
6.              {
7.                  System.out.println("Child Thread:" + i);
8.              }
9.              System.out.println("Exiting child thread");
10.         }
11.     }
12.     class RunnableTest
13.     {
14.         public static void main(String args[])
15.         {
16.             A a1=new A();
17.             Thread t1 = new Thread(a1,"Demo Thread");
18.             t1.start();
```

```

19.         System.out.println("Main thread exiting");
20.     }
21. }

```

Output:

```

Main thread exiting
Child Thread:1
Child Thread:2
Child Thread:3
Child Thread:4
Child Thread:5
Exiting child thread

```

Example12.3

```

1.     class A implements Runnable
2.     {
3.         Thread t;
4.         A()
5.         {
6.             t= new Thread(this, "Demo Thread");
7.             System.out.println("Child thread: "+ t);
8.             t.start();
9.         }
10.        public void run()
11.        {
12.            for (int i=1; i<=5; i++)
13.            {
14.                System.out.println("Child Thread: "+ i );
15.                try
16.                {
17.                    Thread.sleep(500);
18.                }
19.                catch (InterruptedException e)
20.                {

```

```

21.                System.out.println(e);
22.                }
23.            }
24.                System.out.println ("Exiting Child thread");
25.            }
26.        }
27.    class RunnableTest2
28.    {
29.        public static void main (String args[] )
30.        {
31.            A a1=new A();
32.            for (int i=1; i<=5; i++)
33.            {
34.                System.out.println ("Main Thread:"+ i);
35.                try
36.                {
37.                    Thread.sleep (1000);
38.                }
39.                catch(InterruptedException e)
40.                {
41.                    System.out.println("main thread interrupted");
42.                }
43.            }
44.            System.out.println("End of Main thread");
45.        }
46.    }

```

Output:

```

Child thread: Thread[Demo Thread,5,main]
Main Thread:1
Child Thread: 1
Child Thread: 2
Main Thread:2

```

Child Thread: 3
Child Thread: 4
Main Thread:3
Child Thread: 5
Exiting Child thread
Main Thread:4
Main Thread:5
End of Main thread

Extending Thread class:

The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run() method, which is the entry point for the new thread. It must also call start() to begin execution of the new thread.

Example12.4

```
1.      class A extends Thread
2.      {
3.          A()
4.          {
5.              super ("Test Thread");
6.              System.out.println ("Child thread:" + this);
7.              start();
8.          }
9.          public void run ()
10.         {
11.             for (int i=1; i<=5; i++)
12.             {
13.                 System.out.println("Child Thread:"+i);
14.                 try
15.                 {
16.                     sleep(500);
17.                 }
18.                 catch(InterruptedException e)
19.                 {
```



```

20.         System.out.println("Child thread interrupted");
21.             }
22.         }
23.         System.out.println ("Exiting Child thread");
24.     }
25. }
26. class ThreadTest
27. {
28.     public static void main(String args[])
29.     {
30.         new A();
31.         for (int i=1; i<=5; i++)
32.         {
33.             System.out.println("Main Thread: "+i);
34.             try
35.             {
36.                 Thread.sleep(1000);
37.             }
38.             catch(InterruptedExceotion e)
39.             {
40.                 System.out.println("Main thread interrupted");
41.             }
42.         }
43.         System.out.println ("Main thread exiting");
44.     }
45. }

```

Output:

```

Child thread:Thread[Testing Thread,5,main]
Main Thread: 1
Child Thread:1
Child Thread:2
Main Thread: 2

```

Child Thread:3
Child Thread:4
Main Thread: 3
Child Thread:5
Exiting Child thread
Main Thread: 4
Main Thread: 5
Main thread exiting

Note:- If Thread is not assigned any name, it will be something like: Thread-1, Thread-2, Thread-3 etc.

Choosing an Approach:

The thread class defines several methods that can be overridden by a derived class. Out of these methods, the only one that must be overridden is run(). That is, of course the same method required when we implement the Runnable interface.

Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if we will not be overriding any of Thread's other methods, it is probably best simply to implement Runnable interface.

Creating Multiple Threads:

Example12.5

```
1.      class A implements Runnable
2.      {
3.          String name;
4.          Thread t;
5.          A(String threadName)
6.          {
7.              name=threadName;
8.              t=new Thread(this, name);
9.              System.out.println("Child thread:"+t);
10.             t.start();
11.         }
12.         public void run()
13.         {
14.             for(int i=1;i<=5;i++)
15.             {
```

```

16.                System.out.println(t.getName()+ ":" + i);
17.                try
18.                {
19.                    Thread.sleep(1000);
20.                }
21.                catch(InterruptedException e)
22.                {
23.                    System.out.println(e);
24.                }
25.            }
26.            System.out.println(name+ "Exiting");
27.        }
28.    }
29.    class MultiThreadTest
30.    {
31.        public static void main(String args[] )
32.        {
33.            A a1=new A("One");
34.            A a2=new A("Two");
35.            A a3=new A("Three");
36.            try
37.            {
38.                Thread.sleep(10000);
39.            }
40.            catch(InterruptedException e)
41.            {
42.                System.out.println("main thread interrupted");
43.            }
44.            System.out.println ("Main thread exiting");
45.        }
46.    }

```

Output:

Child thread:Thread[One,5,main]
Child thread:Thread[Two,5,main]
Child thread:Thread[Three,5,main]
One:1
Two:1
Three:1
One:2
Two:2
Three:2
One:3
Two:3
Three:3
One:4
Two:4
Three:4
One:5
Two:5
Three:5
OneExiting
TwoExiting
ThreeExiting
Main thread exiting

Imposing some Ordering by using isAlive() and join() Methods:

Often we will want the main thread to finish last. One way to achieve this is to call sleep() within main. This is rather crude (rough) way. Two ways exist to determine whether a thread has finished. First, we can call isAlive() on the thread. This method is defined by Thread, and its general form is:

final boolean isAlive()

The isAlive() method returns true if the thread upon which it is called is still running. It returns false otherwise. While isAlive() is occasionally useful, the method that we will more commonly use to wait for a thread to finish is called join(), shown here:

final void join() throws InterruptedException

This method waits until the thread on which it is called terminates. Additional forms of join() allows us to specify a maximum amount of time that we want to wait for

the specified thread to terminate.

The following example makes use of `join()` to ensure that the main thread is the last to stop. It also demonstrates the `isAlive()` method.

Example12.6

```
1.      class A implements Runnable
2.      {
3.          String name;
4.          Thread t;
5.          A(String threadName)
5.          {
6.              name=threadName;
7.              t=new Thread(this,name);
8.              System.out.println("Child thread:"+t);
9.              t.start();
10.         }
11.         public void run()
12.         {
13.             for(int i=1;i<=5;i++)
14.             {
15.                 System.out.println(name + ":"+ i );
16.                 try
17.                 {
18.                     Thread.sleep(1000);
19.                 }
20.                 catch(InterruptedException e)
21.                 {
22.                     System.out.println(e);
23.                 }
24.             }
25.             System.out.println(name + "Exiting");
```

```

26.         }
27.     }
28.     class DemoJoin
29.     {
30.         public static void main(String args[] )
31.         {
32.             A ob1=new A("One");
33.             A ob2=new A("Two");
34.             A ob3=new A("Three");
35.             System.out.println("Thread One is alive?:"+ob1.t.isAlive());
36.             System.out.println("Thread Two is alive?:"+ob2.t.isAlive());
37.             System.out.println("Thread Three is alive?:"+ob3.t.isAlive());
38.             try
39.             {
40.                 ob3.t.join();
41.                 ob2.t.join();
42.                 ob1.t.join();
43.             }
44.             catch(InterruptedException e)
45.             {
46.                 System.out.println("main thread interrupted");
47.             }
48.             System.out.println("Thread One is alive?:"+ ob1.t.isAlive());
49.             System.out.println("Thread Two is alive?:"+ob2.t.isAlive());
50.             System.out.println("Thread Three is alive?:"+ob3.t.isAlive());
51.             System.out.println ("Main thread exiting");
52.         }
53.     }

```

Output:

Child thread:Thread[One,5,main]

Child thread:Thread[Two,5,main]

Child thread:Thread[Three,5,main]

Thread One is alive?:true
Thread Two is alive?:true
Thread Three is alive?:true
One:1
Two:1
Three:1
One:2
Two:2
Three:2
One:3
Two:3
Three:3
One:4
Two:4
Three:4
One:5
Two:5
Three:5
OneExiting
TwoExiting
ThreeExiting
Thread One is alive?:false
Thread Two is alive?:false
Thread Three is alive?:false
Main thread exiting

Thread Priorities:

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. Higher priority thread will be scheduled first as compared to lower priority thread. A higher priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes from sleeping or waiting on I/O, for example, it will preempt the lower-priority thread.

In theory, threads of equal priority should get equal access to the CPU. But java is designed to work in a wide range of environments. Some of those environment implements multi-tasking fundamentally different than others? For safety, threads that

share the same priority should yield control once in a while.

This ensures that all threads have a chance to run under a non-preemptive OS. In practice, even in non-preemptive environments, most threads inevitably (certainly) encounter some blocking situation, such as waiting for I/O. When this happens, the blocked thread is suspended and other threads can run. For CPU intensive threads, we should make sure that it yields control occasionally, so that other threads can run.

To set a thread's priority, use the `setPriority()` method, which is a member of `Thread`. Its general form is:

final void setPriority(int level)

The value of `level` must be within the range `Thread.MIN_PRIORITY` and `Thread.MAX_PRIORITY`. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify `Thread.NORM_PRIORITY`, which is currently 5. These priorities are defined as final variables within `Thread` class.

We can obtain the current priority setting by calling the `getPriority()` method of `Thread` class, shown here:

final void getPriority()

Implementations of Java have radically different behavior when it comes to scheduling. The windows XP/98/NT/2000 versions work more or less as we would expect. However, other versions may work quite differently. Most of the inconsistencies arise when we have threads that are relying on preemptive behavior, instead of cooperatively giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

The following example demonstrates two threads at different priorities, which do not run on a preemptive platform in the same way as they run on a non-preemptive platform.

Example12.7

```
1.      class Clicker implements Runnable
2.      {
3.          long click = 0;
4.          Thread t;
5.          volatile boolean running=true;
6.          Clicker (int p)
7.          {
8.              t=new Thread (this);
9.              t.setPriority (p);
```



```
10.         }
11.     public void run ()
12.     {
13.         while (running)
14.         {
15.             click++;
16.         }
17.     }
18.     void stop ()
19.     {
20.         running = false;
21.     }
22.     void start ()
23.     {
24.         t.start ();
25.     }
26. }
27. class HiLoPri
28. {
29.     public static void main (String args[] )
30.     {
31.         Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
32.         Clicker hi = new Clicker(Thread.NORM_PRIORITY+1);
33.         Clicker lo = new Clicker(Thread.NORM_PRIORITY-1);
34.         lo.start(); hi.start();
35.         try
36.         {
37.             Thread.sleep (10000);
38.         }
39.         catch (InterruptedException e)
40.         {
41.             System.out.println(e);
```

```

42.         }
43.         lo.stop(); hi.stop();
44.         try
45.         {
46.             hi.t.join();
47.             lo.t.join();
48.         }
49.         catch (InterruptedException e)
50.         {
51.             System.out.println(e);
52.         }
53.         System.out.println("Low : " + lo.click);
54.         System.out.println("High : " + hi.click);
55.     }
56. }

```

Output:

Low : 63660819

High : 2086505403

The higher priority thread gets more than 98% of the CPU time. But threads did context switch, even though neither voluntarily yielded the CPU nor blocked for I/O. When this same program is run under a non-preemptive system, different results will be obtained.

Note:- Variable running is preceded by the keyword volatile to ensure that the value of running is examined each time the following loop iterates.

```

        while (running)
        {
            click++;
        }

```

Without the use of volatile, Java is free to optimize the loop in such a way that a local copy of running is created. The use of volatile prevents the optimization, telling Java that running may change in ways not directly apparent in the immediate code.

Synchronization:

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

Key to synchronization is the concept of the monitor (also called a **semaphore**). A monitor is an object that is used as a mutually exclusive lock. Only one thread can own a monitor at a given time.

When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires. we can synchronize our code in either of two ways:

- (i) Using synchronized methods
- (ii) Using synchronized statements

Using Synchronized Method:

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the synchronized keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait(). To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

Example12.8

Following program illustrates that output generated by three threads gets mixed-up as they are running concurrently.

```
1.      class A
2.      {
3.          void display(String msg)
4.          {
5.              System.out.print("[ " + msg);
6.              try
7.              {
8.                  Thread.sleep(1000);
9.              }
10.         catch(InterruptedException e)
11.         {
12.             System.out.println(e);
13.         }
```

```

14.             System.out.println("");
15.         }
16.     }
17.     class B implements Runnable
18.     {
19.         A obj;
20.         String msg;
21.         Thread t;
22.         B(A obj1, String m)
23.         {
24.             msg = m;
25.             obj = obj1;
26.             t = new Thread(this);
27.             t.start();
28.         }
29.         public void run()
30.         {
31.             obj.display(msg);
32.         }
33.     }
34.     class Synch1
35.     {
36.         public static void main(String args[])
37.         {
38.             A a1 = new A();
39.             B b1 = new B(a1,"Hello");
40.             B b2 = new B(a1,"World");
41.             B b3 = new B(a1,"Matrix");
42.         }
43.     }

```

Output:

[Hello[World[Matrix]

]

]

As we can see, by calling `sleep()`, the `call()` method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time.

This is known as a race condition, because these three threads are racing with each other to complete the method. This example uses `sleep()` to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because we cannot be sure when the context switch will occur. This can cause a program to run right one time and wrong the next time.

To do this, we simply need to precede method `display()`'s definition with the keyword `synchronized`. This prevents either threads from entering the `display()` while another thread is using it.

Example12.9

The problem of mixed-up is solved simply by declaring the method `call()` as a synchronized method.

```
1.      class A
2.      {
3.          synchronized void display(String msg)
4.          {
5.              System.out.print("[ " + msg);
6.              try
7.              {
8.                  Thread.sleep(1000);
9.              }
10.             catch(InterruptedException e)
11.             {
12.                 System.out.println(e);
13.             }
14.             System.out.println("]");
15.         }
16.     }
17.     class B implements Runnable
18.     {
```

```

19.         A obj;
20.         String msg;
21.         Thread t;
22.         B(A obj1, String m)
23.         {
24.             msg = m;
25.             obj = obj1;
26.             t = new Thread(this);
27.             t.start();
28.         }
29.         public void run()
30.         {
31.             obj.display(msg);
32.         }
33.     }
34.     class Synch2
35.     {
36.         public static void main(String args[])
37.         {
38.             A a1 = new A();
39.             B b1 = new B(a1, "Hello");
40.             B b2 = new B(a1, "World");
41.             B b3 = new B(a1, "Matrix");
42.         }
43.     }

```

Output:

[Hello]

[World]

[Matrix]

Note:-Once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, non-synchronized methods on that instance will continue to be callable.

Using Synchronized Statement:

While creating synchronized methods within classes that we create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that we want to synchronize access to objects of a class that was not designed for multi-threaded access. That is, the class does not use synchronized methods. Further, this class was not created by we, but by a third part; and we do not have access to the source code. Thus, we cannot add modifier synchronized to the appropriate methods within the class. How can access to an object of this class by synchronized?

The solution is to put calls to the methods defined by this class inside a synchronized block. The general form of the synchronized statement is:

```
synchronized(object)
{
    //statements to be synchronized
}
```

Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only often the current thread has successfully entered object's monitor.

Example12.10

Alternative version of the preceding example, using a synchronized block within the run()

```
1.      class A
2.      {
3.          void display(String msg)
4.          {
5.              System.out.print("[ " + msg);
6.              try
7.              {
8.                  Thread.sleep(1000);
9.              }
10.         catch(InterruptedException e)
11.         {
12.             System.out.println(e);
13.         }
14.         System.out.println("]");
15.     }
16. }
```

```
17.     class B implements Runnable
18.     {
19.         A obj;
20.         String msg;
21.         Thread t;
22.         B(A obj1, String m)
23.         {
24.             msg = m;
25.             obj = obj1;
26.             t = new Thread(this);
27.             t.start();
28.         }
29.         public void run()
30.         {
31.             synchronized(obj)
32.             {
33.                 obj.display(msg);
34.             }
35.         }
36.     }
37.     class Synch3
38.     {
39.         public static void main(String args[])
40.         {
41.             A a1 = new A();
42.             B b1 = new B(a1, "Hello");
43.             B b2 = new B(a1, "World");
44.             B b3 = new B(a1, "Matrix");
45.         }
46.     }
```

Output:-

[Hello]

[World]

[Matrix]

Multiple Threads Invoking same Method on Different Objects:

Example12.11

Threads can call the same synchronized instance method on different objects of same class concurrently as each object has a different lock. This will again lead to mixed-up output.

```
1.     class A
2.     {
3.         synchronized void display(String msg)
4.         {
5.             System.out.print("[ " + msg);
6.             try
7.             {
8.                 Thread.sleep(1000);
9.             }
10.            catch(InterruptedException e)
11.            {
12.                System.out.println(e);
13.            }
14.            System.out.println("]");
15.        }
16.    }
17.    class B implements Runnable
18.    {
19.        A obj;
20.        String msg;
21.        Thread t;
22.        B(A obj1, String m)
23.        {
24.            msg = m;
25.            obj = obj1;
26.            t = new Thread(this);
27.            t.start();
```

```

28.         }
29.     public void run()
30.     {
31.         obj.display(msg);
32.     }
33. }
34. class Synch4
35. {
36.     public static void main(String args[])
37.     {
38.         A a1 = new A();
39.         A a2 = new A();
40.         A a3 = new A();
41.         B b1 = new B(a1, "Hello");
42.         B b2 = new B(a2, "World");
43.         B b3 = new B(a3, "Matrix");
44.     }
45. }

```

Output:

```

    [Hello[World[Matrix]
    ]
    ]

```

Using Static Synchronized Method:

Example12.12

The output does not get mixed-up in the following example although we are calling display() method on different objects. The reason is that method display() is static and hence the lock is obtained for the class A as a whole and not on the individual objects.

```

1.     class A
2.     {
3.         static synchronized void display(String msg)
4.         {
5.             System.out.print("[ " + msg);

```

```
6.         try
7.         {
8.             Thread.sleep(1000);
9.         }
10.        catch(InterruptedException e)
11.        {
12.            System.out.println(e);
13.        }
14.        System.out.println("");
15.    }
16. }
17. class B implements Runnable
18. {
19.     A obj;
20.     String msg;
21.     Thread t;
22.     B(A obj1, String m)
23.     {
24.         msg = m;
25.         obj = obj1;
26.         t = new Thread(this);
27.         t.start();
28.     }
29.     public void run()
30.     {
31.         obj.display(msg);
32.     }
33. }
34. class Synch5
35. {
36.     public static void main(String args[])
37.     {
```

```

38.          A a1 = new A();
39.          A a2 = new A();
40.          A a3 = new A();
41.          B b1 = new B(a1,“Hello”);
42.          B b2 = new B(a2,“World”);
43.          B b3 = new B(a3,“Matrix”);
44.          }
45.      }

```

Output:-

```

[Hello]
[World]
[Matrix]

```

Inter-thread Communication:

It is discussed earlier, multi-threading replaces event loop programming by dividing our tasks into discrete and logical units. Threads also proved a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken.

This wastes the CPU cycles. For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interacting, suppose that the producer has to wait until the consumer has finished before it generates more data. In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on.

To avoid polling, Java includes an elegant (smart) inter-process communication mechanism via the wait(), notify(), and notifyAll() methods. These methods are implemented as final methods in Object class, so all classes have them. All three methods can be called only from within a synchronized context. There are certain rules for using these methods.

Wait() tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify().

Notify() wakes up one thread (normally first thread) that called wait on the same object.

notifyAll() wakes up all the threads that called wait() on the same object. Normally the highest priority thread will run first.

These methods are declared within Object class, as shown here:

final void wait() throws InterruptedException

final void notify()

final void notifyAll()

Example 12.13

The following sample program incorrectly implements a simple form of the producer/consumer problem. It consists four classes: Q, the queue that we are trying to synchronize; Producer, the threaded object that is producing queue entries; Consumer, the threaded object that is consuming g queue entries; and PC, the tiny class that creates the sing Q, Producer, and Consumer.

```
1.      class Q
2.      {
3.          int n;
4.          synchronized int get()
5.          {
6.              System.out.println("Got:"+n);
7.              return n;
8.          }
9.          synchronized void put(int n)
10.         {
11.             this.n=n;
12.             System.out.println("Put:"+n);
13.         }
14.     }
15.     class Producer implements Runnable
16.     {
17.         Q q1;
18.         Producer(Q q2)
19.         {
20.             q1=q2;
21.             new Thread(this,"Producer").start();
22.         }
23.         public void run()
24.         {
25.             int i=0;
```

```

26.             while(true)
27.             {
28.                 q1.put(++i);
29.             }
30.         }
31.     }
32.     class Consumer implements Runnable
33.     {
34.         Q q1;
35.         Consumer(Q q2) {
36.             q1=q2;
37.             new Thread(this,"Consumer").start();
38.         }
39.         public void run()
40.         {
41.             while (true)
42.             {
43.                 q1.get();
44.             }
45.         }
46.     }
47. }
48. class PC
49. {
50.     public static void main(String args[])
51.     {
52.         Q q1=new Q();
53.         new Producer(q1);
54.         new Consumer(q1);
55.         System.out.println("Press Control-C to stop");
56.     }
57. }

```

Although the put() and get() methods on Q are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the

same queue value twice. Thus we may get erroneous output as shown below:

Output: Press Control-C to stop

Put :1

Put :2

Put:3

Put:4

Put:5

Put:6

Put:7

Got:7

Got:7

Got:7

Got:7

Got:7

Got:7

Got:7

Got:7

Put:8

Put:9

Put:10

Put:11

Put:12

Put:13

Put:14

Put:15

Got:15

Got:15

Got:15

Got:15

Got:15

Got:15

Got:15

Example12.14

The proper way to write this program in Java is to use `wait()` and `notify()` to signal in both directions, as shown here:

1. `class Q`
2. `{`

```
3.         int n;
4.         boolean valueSet=false;
5.         synchronized int get ()
6.         {
7.             if(!valueSet)
8.             {
9.                 try
10.                {
11.                    wait();
12.                }
13.            catch (InterruptedException e)
14.            {
15.                System.out.println (e);
16.            }
17.        }
18.        System.out.println("Got :" + n);
19.        valueSet = false;
20.        notify();
21.        return n;
22.    }
23.    synchronized void put (int n)
24.    {
25.        if(valueSet)
26.        {
27.            try
28.            {
29.                wait();
30.            }
31.        catch(InterruptedException e)
32.        {
33.            System.out.println(e);
34.        }
```



```
35.         }
36.         this.n=n;
37.         valueSet =true;
38.         System.out.println("Put :” + n);
39.         notify ();
40.     }
41. }
42. class Producer implements Runnable
43. {
44.     Q q1;
45.     Producer (Q q2)
46.     {
47.         q1 = q2;
48.         new Thread(this, “Producer”).start ();
49.     }
50.     public void run ( )
51.     {
52.         int i = 0;
53.         while (true)
54.         {
55.             q1.put (++i);
56.         }
57.     }
58. }
59. class Consumer implements Runnable
60. {
61.     Q q1;
62.     Consumer (Q q2)
63.     {
64.         q1 =q2;
65.         new Thread (this, “consumer”).start ( );
66.     }
```

```
67.         public void run ( )
68.         {
69.             while (true)
70.             {
71.                 q1.get ( );
72.             }
73.         }
74.     }
75.     class PC2
76.     {
77.     public static void main (String args [])
78.     {
79.     System.out.println("Press Control-C to stop");
80.         Q q1 = new Q( );
81.         new Producer(q1);
82.         new Consumer(q1);
83.     }
84.     }
```

Output:

Press Control-C to stop

Put:1

Got:1

Put:2

Got:2

Put:3

Got:3

Put:4

Got:4

Put:5

Got:5

Deadlock:

A special type of error that we need to avoid related specifically to multi-tasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons:

- (i) In general, it occurs only rarely when the two threads time-slice in just the right way.
- (ii) It may involve more than two threads and two synchronized objects.

Example 12.15

```
1.     class A
2.     {
3.         synchronized void foo(B b1)
4.         {
5.             System.out.println("Method foo called and then blocked");
6.             try
7.             {
8.                 Thread.sleep(1000);
9.             }
10.            catch(InterruptedException e)
11.            {
12.                System.out.println(e.getMessage());
13.            }
14.            System.out.println("Trying to call b1.last()");
15.                b1.last();
16.            }
17.        synchronized void last()
18.        {
19.            System.out.println("This will not be printed");
20.        }
21.    }
```

```
22.     class B
23.     {
24.         synchronized void bar(A a1)
25.         {
26.             System.out.println("Method bar called and then blocked");
27.             try
28.             {
29.                 Thread.sleep(1000);
30.             }
31.             catch(InterruptedException e)
32.             {
33.                 System.out.println(e.getMessage());
34.             }
35.             System.out.println("Trying to call a1.last()");
36.             a1.last();
37.         }
38.         synchronized void last()
39.         {
40.             System.out.println("This will not be printed");
41.         }
42.     }
43.     class DeadLock implements Runnable
44.     {
45.         A a1= new A();
46.         B b1 = new B();
47.         DeadLock()
48.         {
49.             Thread t = new Thread(this);
50.             t.start();
51.             a1.foo(b1);
52.         }
```

```

53.         public void run()
54.         {
55.             b1.bar(a1);
56.         }
57.         public static void main(String args[])
58.         {
59.             new DeadLock();
60.         }
61.     }

```

Output:

```

Method foo called and then blocked
Method bar called and then blocked
Trying to call b1.last()
Trying to call a1.last()

```

Suspending, Resuming and Stopping Threads in Java 1.1 and earlier:

Prior to Java 2, suspend() and resume() are defined by Thread class to pause and restart the execution of a thread. They have the form shown below.

```

final void suspend( )
final void resume( )

```

Example12.16:

```

1.     class A implements Runnable
2.     {
3.         String name;
4.         Thread t;
5.         A(String threadname)
6.         {
7.             name = threadname;
8.             t = new Thread(this, name);
9.             System.out.println("Child Thread: " + t);
10.            t.start();
11.        }

```

```
12.         public void run()
13.         {
14.             for(int i=1; i<=30;i++)
15.             {
16.                 System.out.println(name + ": " + i);
17.                 try
18.                 {
19.                     Thread.sleep(100);
20.                 }
21.                 catch(InterruptedException e)
22.                 {
23.                     System.out.println(e);
24.                 }
25.             }
26.             System.out.println(name + " exiting");
27.         }
28.     }
29.     class SuspendResume
30.     {
31.         public static void main(String args[])
32.         {
33.             A ob1 = new A("One");
34.             A ob2 = new A("Two");
35.             try
36.             {
37.                 Thread.sleep(1000);
38.                 ob1.t.suspend();
39.                 System.out.println("Suspending Thread one");
40.                 Thread.sleep(1000);
41.                 ob1.t.resume();
42.                 System.out.println("Resuming Thread one");
43.                 ob2.t.suspend();
```

```

44.         System.out.println("Suspending Thread two");
45.             Thread.sleep(1000);
46.             ob2.t.resume();
47.         System.out.println("Resuming Thread two");
48.             }
49.     catch(InterruptedException e)
50.     {
51.         System.out.println(e);
52.     }
53.     try
54.     {
55.         System.out.println("Waiting for threads to finish");
56.             ob1.t.join();
57.             ob2.t.join();
58.         }
59.     catch(InterruptedException e)
60.     {
61.         System.out.println(e);
62.     }
63. }
64. }

```

Output:

Output:

Child Thread: Thread[One,5,main]

Child Thread: Thread[Two,5,main]

One: 1

Two: 1

Two: 2

One: 2

Two: 3

One: 3

Two: 4

One: 4

One: 5

Two: 5

Two: 6

One: 6

One: 7

Two: 7

Two: 8

One: 8

One: 9

Two: 9

Two: 10

One: 10

Suspending Thread one

Two: 11

Two: 12

Two: 13

Two: 14

Two: 15

Two: 16

Two: 17

Two: 18

Two: 19

Two: 20

Resuming Thread one

Suspending Thread two

One: 11

One: 12

One: 13

One: 14

One: 15

One: 16

One: 17

One: 18

One: 19

One: 20

Resuming Thread two

One: 21

Two: 21

One: 22

Two: 22

One: 23

Two: 23

One: 24

Two: 24

One: 25

Two: 25

One: 26

Two: 26

One: 27

Two: 27

One: 28

Two: 28

One: 29

Two: 29

One: 30

Two: 30

One exiting

Two exiting

Waiting for threads to finish

Note:-The thread class also defines a method called stop() that stops a thread. Its signature is shown here:

final void stop()

Once a thread has been stopped, it cannot be restarted using resume() method.

Suspending, Resuming and stopping threads using Java 2

Example12.17:

```
1.      class A implements Runnable
2.      {
3.          String name;
4.          Thread t;
5.          boolean suspendFlag;
6.          A(String threadname)
7.          {
8.              name = threadname;
9.              t = new Thread(this, name);
10.             System.out.println("Child Thread: " + t);
11.             suspendFlag = false;
12.             t.start();
13.         }
14.         void mysuspend()
15.         {
16.             suspendFlag = true;
17.         }
18.         synchronized void myresume()
19.         {
20.             suspendFlag = false;
21.             notify();
22.         }
23.         public void run()
24.         {
25.             for(int i=1;i<=15;i++)
26.             {
27.                 System.out.println(name + ": " + i);
28.                 try
29.                 {
```

```
30.         Thread.sleep(200);
31.             }
32.         catch(InterruptedException e)
33.             {
34.             System.out.println(e);
35.             }
36.         synchronized(this)
37.             {
38.             if(suspendFlag)
39.                 {
40.                 wait();
41.                 }
42.             }
43.         }
44.         System.out.println(name+ "exiting");
45.     }
46. }
47. class SuspendResume1
48. {
49.     public static void main(String args[])
50.     {
51.         A ob1 = new A("One");
52.         A ob2 = new A("Two");
53.         try
54.         {
55.             Thread.sleep(1000);
56.             ob1.mysuspend();
57.             System.out.println("Suspending Thread one");
58.             Thread.sleep(1000);
59.             ob1.myresume();
60.             System.out.println("Resuming Thread one");
61.             ob2.mysuspend();
```

```
62.         System.out.println("Suspending Thread two");
63.         Thread.sleep(1000);
64.         ob2.myresume();
65.         System.out.println("Resuming Thread two");
66.     }
67.     catch(InterruptedException e)
68.     {
69.         System.out.println(e);
70.     }
71.     try
72.     {
73.         System.out.println("Waiting for threads to finish");
74.         ob1.t.join();
75.         ob2.t.join();
76.     }
77.     catch(InterruptedException e)
78.     {
79.         System.out.println(e);
80.     }
81.     System.out.println("main exiting");
82.     } }
```

Output:

Child Thread: Thread[One,5,main]

Child Thread: Thread[Two,5,main]

One: 1

Two: 1

One: 2

Two: 2

One: 3

Two: 3

One: 4

Two: 4

One: 5

Two: 5

Suspending Thread one

Two: 6

Two: 7

Two: 8

Two: 9

Two: 10

Resuming Thread one

One: 6

Suspending Thread two

One: 7

One: 8

One: 9

One: 10

Resuming Thread two

Two: 11

Waiting for threads to finish

One: 11

Two: 12

One: 12

Two: 13

One: 13

Two: 14

One: 14

Two: 15

One: 15

Two exiting

One exiting

main exiting



CHAPTER

∞ 13 ∞

(Modifiers / Visibility modes)

Introduction-

Modifiers are Java keywords that give the compiler information about the nature of code, data, classes or interfaces. For example, we have been using visibility modifiers `public`, `private`, `protected`, `package public` etc. to specify the visibility of class members. Beside visibility we have also used the modifier `static`, `final` and `abstract`.

For the purpose of clear understanding, modifier can be categorized as:

1. *Accessibility modifiers for top-level classes and interfaces.*
2. *Member accessibility/visibility modifiers for classes.*
3. *Member accessibility/visibility modifiers for interfaces.*
4. *Other modifiers for top-level classes.*
5. *Other modifiers for top-level interfaces.*
6. *Other modifiers for interface members.*
7. *Other modifiers for class members.*

1. Accessibility modifiers for top-level classes and interfaces:

- *public*
- *default (package) accessibility*

2. Member accessibility/visibility modifiers for classes:

By specifying member accessibility modifiers a class can control what information is accessible to clients (i.e. other classes). These modifiers help a class to define a contract so that clients know exactly what services are offered by the class.

Accessibility/visibility of members can be one of the following:

- ***public***
- ***protected***
- ***default (also called package accessibility)***
- ***private***

Note:-Member accessibility modifiers only has meaning if the class (or one of its subclasses) is accessible to the client. Also note that one accessibility modifier can be specified for a member.

The discussion applies to both instance and static members of classes.

3. Member accessibility/visibility modifiers for interfaces:

The only member accessibility/visibility modifier that can be used with data members and methods of an interface is public.

The public is also the implicit accessibility/visibility modifier for interface members i.e. the members are always implicitly assumed to be public even if we do not use the modifier public.

4. Other modifiers for top-level classes:

Beside visibility modifiers, we can also use following modifiers before a top-class:

(a) abstract

A class can be specified with the keyword abstract to indicate that it cannot be instantiated. A class containing abstract method must be declared as abstract otherwise it won't compile. A class not containing abstract method can also be declared abstract. Such a class can serve as a base class for a number of sub-classes.

(b) final

A class can be declared final to indicate that it cannot be extended. The final class marks the lower boundary of its implementation inheritance hierarchy. Only a class whose definition is complete can be declared final. A class cannot be both final and abstract at the same time.

Here are few important characteristics of the final class.

All the methods of a final class are also final i.e. they have the concrete implementation and can not be over-ridden.

- 👉 Some type checks become faster with final classes. In fact, many type checks

become compile time checks and errors can be caught earlier. If the compiler encounters a reference to a final class, it knows that the object referred to is exactly of that type.

- ✦ The compiler is able to perform certain code optimizations for final methods, because certain assumptions can be made about such members.
- ✦ When a non-final method is invoked, the run time system determines the actual class of the object, binds the method invocation to the correct implementation of the method for that type, and then invokes the implementation.
- ✦ In case of a final method we are sure that type sub-class can not override the method so the binding is done at the compile time as in case of private and static methods, which would definitely improve the performance.
- ✦ In case of a final method the compiler may replace an invocation with the actual body of the method like Macro. This mechanism is known as 'inlining'. In C++, we have the option of declaring a function as inline (although final decision is taken by the compiler) but in Java, the compiler takes the decision.

5. Other modifiers for top-level interfaces:

(a) abstract

This is the only modifier other than visibility modifiers, which can be used with interface.

Interfaces just specify the method prototypes and not any implementation: they are, by their nature, implicitly abstract. (i.e. they can not be instantiated). We can declare an interface as abstract but it is redundant as interface is always abstract.

6. Other modifiers for interface members:

Other modifiers for data members:

Beside visibility modifier public, we can also use modifiers static and final. Although we can use these modifiers but it is redundant as all data members are implicitly public, static and final.

Other modifiers for methods:

Beside visibility modifier public, we can also use modifier abstract. Although we can use abstract modifier but it is redundant as all methods are implicitly public, and abstract.

7. Other modifiers for class members:-

Certain characteristics of fields and/or methods can be specified in their declarations by the following keywords:

1. **static**
2. **final**
3. **abstract**

4. **synchronized**
5. **native**
6. **transient**
7. **volatile**

A. static modifier:

The static members belong to the class in which they are declared, and are not part of any instance of the class. Depending on the accessibility modifiers of the static members in a class, client can access these by using the class name or through object references of the class.

static variables(also called class variables):-

These variables only exist in the class they are defined in. When the class is loaded, static variables are initialized to their default values, if no explicit initialization expression is specified.

The static member variables are often used when tracking global information about the instances.

static methods:

These are also known as class methods. A static method in a class can directly access other static members in the class. It cannot access instance (i.e. non-static) members of the class, as there is no notion of an object associated with a static method.

However, note that a static method in a class can always use a reference of the class's type to access its members regardless of whether these members are static or not. A typical static method might perform some task on behalf of the whole class and/or objects of the class.

An instance in a subclass cannot override a static method in the superclass. The compiler will flag this with an error (means a static method in super class cannot be overridden by non static methods in subclass). A static method is class specific and not part of any object, while overriding , methods are invoked on the behalf of objects of the subclass. However, a static method in a subclass can hide a static method in the superclass.

Methods declared as static have several restrictions:

- They can only call other static methods.
- They can only access static data members.
- They cannot refer to this or super in anyway

static initialization block:

A java class can have one or more static initialization block. The syntax of the static

initialization block is as follows:

```
static
{
code
}
```

A static initialization block can be thought of as a static method, which is invoked implicitly/automatically as soon as the class is loaded.

The static block can be useful in the following situations:

- Dynamic initialization of static variables.
- For performing one time activity at the time of loading class. For example, loading jdbc driver class.
- Loading small database/configuration files in Hashtable, HashMap, Properties etc.

B. final modifier:

The modifier final can be used with

- Local variables
- Instance variables/data members
- Static variables/data members
- Instance methods

final variables:

A final variable is normally initialized at the time of declaration and its value can not be modified after assigning once. Here are few important points related to final variables:

- A final variable is a constant, despite being called a variable. Its value cannot be changed once it has been initialized. This applies to instance, static and local variables, including parameters that are declared final.
- The final is the only modifier applicable to local variables or formal parameters.
- A final variable of a primitive data type cannot change its value once it has been initialized.
- A final variable of a reference type cannot change its reference value once it has been initialized, but the state of the object it denotes can still be changed.
- Normally a final variable is initialized at the time of declaration but it is not must. Such a final variable is also called blank final variable, and must be initialized

once before it is being used.

- The final static variables are commonly used to define named constants, for example Integer.MAX_VALUE, which is the maximum int value.

Local final variables:-

Example 13.1

The following example illustrates that local final variable can be left blank as discussed above but must be initialized before first use. The variable x is initialized just before displaying its value on the monitor.

```
1.      class FinalTest1 // make use of blank final
2.      {
3.          public static void main(String args[])
4.          {
5.              final int x;          //initialization at declaration time not must
6.                  x = 50;
7.                  System.out.println(x);
8.          }
9.      }
```

Output: 50

Example13.2

The following will not compile as final variable is initialized twice.

```
      class FinalTest2 // make use of blank final
1.      {
2.          public static void main(String args[ ])
3.          {
4.              final int x;
5.                  x=50;
6.                  System.out.println(x);
7.                  x = 60;
8.          }
9.      }
```

Instance final variables:-

An instance final variable can be left blank but must be initialized before obtaining a

reference of an object of the class containing final variable. So if we do not initialize at the time of declaration then the other places at which final variables can be initialized are:

- Constructor
- Initialization block

Example 13.3

The following program will not compile, as final variable is not initialized. The concept of initialization by default value is not applicable to final variables as they can not be modified after initialization.

```
1.          class FinalTest3 // make use of blank final
2.          {
3.      final int x; // No default initialization for final instance variables
4.          }
5.          class MyClass
6.          {
7.              public static void main(String args[])
8.              {
9.                  FinalTest3 f=new FinalTest3();
10.                 System.out.println(f.x);
11.             }
12.         }
```

Output:

(Compile time error): variable x might not have been initialized.

Example 13.4

The following program will also not compile, as final variable is initialized after obtaining a reference.

```
1.          class FinalTest4
2.          {
3.              final int x;
4.          }
5.          class MyClass
6.          {
```

```

7.         public static void main(String args[])
8.         {
9.             FinalTest4 f=new FinalTest4();
10.            f.x = 10;
11.            System.out.println(f.x);
12.        }
13.    }

```

Example 13.5

The following example illustrates that a blank final instance variable can be initialized in the constructor.

```

1.         class FinalTest5
2.         {
3.             final int x;
4.             public static void main(String args[])
5.             {
6.                 FinalTest5 f=new FinalTest5();
7.                 System.out.println(f.x);
8.             }
9.             FinalTest5()
10.            {
11.                x=10;
12.            }
13.        }

```

Output:

10

Example 13.6

The following program illustrates that a blank final variable can be initialized with a dynamic value and can be referred with this keyword. So different objects of the same class can have different values of the final variable.

```

1.         class FinalTest6
2.         {
3.             final int x;

```

```

4.         FinalTest6(int x1)
5.         {
6.             x = x1;
7.         }
8.         public static void main(String args[])
9.         {
10.            FinalTest6 f1=new FinalTest6(10);
11.            FinalTest6 f2=new FinalTest6(20);
12.            System.out.println(f1.x);
13.            System.out.println(f2.x);
14.        }
15.    }

```

Output:

```

10
20

```

Initialization block:-

A java class can have one or more initialization blocks. The syntax of the initialization block is as follows:

```

{
    code
}

```

A initialization block can be thought if as a constructor, which is invoked implicitly/automatically as soon as the object is create. In fact if a class contains initialization blocks them they are executed in the order of definition from top to bottom before any constructor.

Example 13.7:

```

1.     class FinalTest7
2.     {
3.         final int x;
4.         public static void main(String args[])
5.         {

```

```

6.          FinalTest7 f=new FinalTest7();
7.          System.out.println(f.x);
8.          }
9.          {
10.         x=20;
11.         }
12.     }

```

The initialization block can be useful in the following situations:

- Dynamic initialization of instance variables.
- For defining code which is common to all constructors. This will increase the code reusability and reduce maintenance.

A static final variables:

A static final variables can be left blank, but must be initialized before class is available to the program. So if we do not initialize at the time of declaration then the only place at which static final variable can be initialized is:

- **static Initialization block**

Example 13.8

The following example illustrates that the blank static final variable can be initialized in the static block.

```

1.     class FinalTest8
2.     {
3.         static final int x;
4.         public static void main(String args[])
5.         {
6.             System.out.println(FinalTest8.x);
7.         }
8.         static
9.         {
10.            x=20;
11.        }

```

12. }

Output:

20

final methods:-

- ✦ A final method in a class is complete (i.e. has an implementation) and can not be overridden in any subclass. Subclasses are thus restricted in changing the behavior of the method.
- ✦ The compiler is able to perform certain code optimizations for final methods, because certain assumptions can be made about such members. When a non-static method is invoked, the run time system determines the actual class of the object, binds the method invocation to the correct implementation of the method for that type, and then invokes the implementation.
- ✦ In case of a final method we are sure that sub-class can not override the method so the binding is done at the compile time as in case of private and static methods, which would definitely improves the performance.
- ✦ In case of a final method the compiler may replace an invocation with the actual body of the method like Macro. This mechanism is known as 'inlining'. In C++, we have the option of declaring a function as inline (although final decision is taken by the compiler) but in Java, the compiler takes the decision.

C. abstract modifier:-

The abstract modifier can be used only with instance methods. An abstract method does not have an implementation, that is, no method body is defined for an abstract method, only the method prototype is provided in the class definition.

Its class is also abstract (i.e., incomplete) and must be explicitly declared as abstract. Subclasses of an abstract class must then provide the method implementation; otherwise, they are also abstract.

Here are some important points related to abstract methods:

- ✦ An abstract method cannot be declared private. This is obvious as we can not override an private method, while in case of an abstract method body is always provided in the sub-class by over-riding the abstract method.
- ✦ Only an instance method can be declared abstract since static methods cannot be overridden, declaring an abstract static method would make no sense and will result in compilation error.
- ✦ A final method cannot be abstract and vice-versa.
- ✦ The keyword abstract cannot be combined with any non-accessibility modifiers for methods.
- ✦ Methods specified in an interface are implicitly abstract, as only the method prototypes are defined in an interface.

D. synchronized modifier:-

The synchronized keyword can be used with methods and code blocks. Several threads can execute simultaneously in a program. They might try to execute several methods on the same object simultaneously in a program. They might try to execute several methods on the same object simultaneously. If it is desired that only one thread at a time can execute a method in the object, the methods can be declared synchronized. Their execution is mutually exclusive among all threads.

At any given time, at most one thread can be executing a synchronized method on any given object, at most one thread can be executing a synchronized method on an object. This discussion also applies to static synchronized methods of a class.

Note- The synchronized modifier does not apply to classes or member variables.

E. native modifier:-

The native modifier can refer only to methods. Like the abstract keyword, native indicates that the body of a method is to be found elsewhere. The body of a native method is entirely outside the JVM, in a library. Native code is written in a non-Java language, typically C or C++, and compiled for a single target machine type. Thus Java's platform independence is violated. People who port Java to new platforms implement extensive native code to support GUI components, network communication, and a broad range of other platform-specific functionality. However, it is rare for the application and applet programmers to need to write native code.

F. transient modifier:-

Objects can be stored using serialization. Serialization transforms objects into an output format that is conducive for storing objects. Objects can later be retrieved in the same state as when they were serialized, meaning that all fields included in the serialization will have the same values as at the time of serialization. Such objects are said to be persistent.

A field can be specified as transient in the class declaration, indicating that its value should not be saved when objects of the class are written to persistent storage. class Experiment implements Serializable-

```
{
    transient int currentTemperature;    //transient
    double mass;                        //persistent value
}
```

Specifying the transient modifier for static variables is redundant and therefore, discouraged. The static variables are not part of the persistent state of a serialized object.

G. volatile modifier

During execution, compiled code might cache the value of fields for efficiency reasons. Since multiple threads can access the same field, it is vital that caching is not allowed to cause inconsistencies when reading and writing the value in the field. The

volatile modifier can be used to inform the compiler that it should not attempt to perform optimization on the field, which could cause unpredictable results when the field is accessed by multiple threads.



CHAPTER

∞ 14 ∞

(Wrapper Class)

Introduction-

The primitive data types in java are not objects. If we want to use these data types as objects, then we will have to use wrapper classes for each of these primitive data types provided in java.lang package.

There are many built-in classes, which cannot handle primitive data types as they deal only with objects. One such class is Vector, which is used to store a collection of objects. We can not use the Vector class to directly store the collection of elements of a primitive data typed. But we can do so by storing the objects of wrapper classes, which correspond to the primitive data types.

The Java has wrapper class corresponding to each of the primitive data type as shown in the following table:

--

Primitive Data Type	Wrapper Class
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Number class:

The abstract class Number defines super-class that is implemented by the classes that wrap the numeric type byte, short, int, long, float, and double. Number class has abstract methods that return the value of the object in each of the different number formats. These methods are:

byte byteValue()

Returns the value of the specified number as a byte.

short shortValue()

Returns the value of the specified number as a short.

abstract int intValue()

Returns the value of the specified number as an int.

abstract long longValue()

Returns the value of the specified number as a long.

abstract float floatValue()

Returns the value of the specified number as a float.

abstract double doubleValue()

Returns the value of the specified number as a double.

Note- The values returned by `byteValue()`, `shortValue()`, `intValue()`, `longValue()`, `floatValue()` and `doubleValue()` methods may involve rounding or truncation.

Example 14.1

The following example demonstrates how rounding and truncation takes place when invoking methods of class `Number`.

```
1.     class Wrapper
2.     {
3.         public static void main(String args[ ])
4.         {
5.             Integer iObj = new Integer(128);
6.             System.out.println(iObj.byteValue()); //truncation
7.             Long lObj = new Long(123456789123456789L);
8.             System.out.println(lObj);
9.             System.out.println(lObj.doubleValue());
10.            Float fObj = new Float(3.99f);
11.            System.out.println(fObj.intValue()); truncation
12.        }
13.    }
```

Output:

```
-128
123456789123456789
1.23456789123456784E17
3
```

Converting Primitive Numbers to Objects using Constructors of Wrappers Classes and Converting Numeric Objects back to Primitive Numbers:-

Example 14.2

The following example demonstrates how primitives can be wrapped in objects and how they can be converted back to primitives.

```
1.     class Convert
```

```
2.     {
3.         public static void main(String args[])
4.         {
5. System.out.println("Converting primitive numbers to objects " + "using constructors");
6.
7.             byte b = 105;
8.             Byte bObj = new Byte(b);
9. System.out.println(bObj); //toString()
10.            short s = 2015;
11.            Short sObj = new Short(s)
12.            System.out.println(sObj);
13.            int i=32717;
14.            Integer iObj = new Integer(i);
15.            System.out.println(iObj);
16.            long l = 234543335565675L
17.            Long lObj = new Long(l);
18.            System.out.println(lObj);
19.            float f = 3.1415f;
20.            Float fObj = new Float(f);
21.            System.out.println(fObj);
22.            double d = 3.1415;
23.            Double dObj = new Double(d);
24.            System.out.println(dObj);
25. System.out.println("Converting numeric objects to primitive numbers");
26.            byte b1 = bObj.byteValue();
27.            short s1 = sObj.shortValue();
28.            int i1 = iObj.intValue();
29.            long l1 = lObj.longValue();
30.            float f1 = fObj.floatValue();
31.            double d1 = dObj.doubleValue();
32.            System.out.println(b1);
33.            System.out.println(s1);
```

```

34.          System.out.println(i1);
35.          System.out.println(l1);
36.          System.out.println(f1);
37.          System.out.println(d1);
38              }
39.      }

```

Output:

Converting primitive numbers to objects using constructor

105

2015

32717

234543335565675

3.1415

3.1415

Converting object to primitive numbers

105

2015

32717

234543335565675

3.1415

3.1415

Converting Primitive Numbers to Strings using toString() static method of the corresponding Wrapper Class

Example 14.3:

```

1.      class ConvertPrimitiveToString
2.      {
3.          public static void main(String args[ ])
4.          {
5.              System.out.println("Converting primitive numbers to String " +
6.              "using toString() static method of corresponding wrapper class:");
7.              byte b = 105;
8.              String str=Byte.toString(b);

```

```

9.          System.out.println(str);
10.         short s=303;
11.         str = Short.toString(s);
12.         System.out.println(str);
13.         int i=100;
14.         str = Integer.toString(i);
15.         System.out.println(str);
16.         long l=4544444444444l;
17.         str = Long.toString(l);
18.         System.out.println(str);
19.         float f=3.444f;
20.         str=Float.toString(f);
21.         System.out.println(str);
22.         double d=3.44444;
23.         str=Double.toString(d);
24.         System.out.println(str);
25.     }
26 }

```

Output:

Converting primitive numbers to String using toString() static method of corresponding wrapper class:

105

303

100

4544444444444

3.444

3.44444

Converting Numeric Objects to Strings using toString() method of the corresponding Wrapper Class:-

Example 14.4

```
1.      class ObjectToStringDemo
```

```

2.      {
3.          public static void main(String args[ ])
4.          {
5.              System.out.println("Converting object numbers to Strings using" +
        "toString() method of corresponding wrapper class:");
7.              byte b=103;
8.              Byte bObj = new Byte(b);
9.              String str=bObj.toString();
10.             System.out.println(str);
11.             short s=203;
12.             Short sObj=new Short(s);
13.             str = sObj.toString();
14.             System.out.println(str);
15.             Integer iObj = new Integer(32000);
16.             str = iObj.toString();
17.             System.out.println(str);
18.             str = new Long(454444444444444l).toString();
19.             System.out.println(str);
20.             str = new Float(3.1444f).toString();
21.             System.out.println(str);
22.             str = new Double(4.1444).toString();
23.             System.out.println(str);
24.         }
25.     }

```

Output:

Converting object numbers to Strings using toString() method of corresponding wrapper class:

103

203

32000

454444444444444

3.1444

4.1444

Converting String Objects(Numeric Strings) to Numeric Objects using the static valueOf() method of the corresponding Wrapper Class

Example 14.5

```
1.      class StringToNumericObjectDemo
2.      {
3.          public static void main(String args[])
4.          {
5.              String str="30";
6.              String str2="30.333";
7.              Byte bObj=Byte.valueOf(str);
8.              System.out.println(bObj);
9.              //Byte bObj1 = new Byte(str2);          //NumberFormatException
10.             Short sObj = Short.valueOf(str);
11.             System.out.println(sObj);
12.             Integer iObj=Integer.valueOf(str);
13.             System.out.println(iObj);
14.             Long lObj=Long.valueOf("344324232432");
15.             System.out.println(lObj);
16.             Float fObj=Float.valueOf("3.333");
17.             System.out.println(fObj);
18.             Double dObj=Double.valueOf(str2);
19.             System.out.println(dObj);
20.         }
21.     }
```

Output:

30

30

30

3.44324232432

3.33

30.333

Note:-All of the valueOf() methods throw “NumberFormatException” if the string does not contain a parsable number.

Converting string Objects (Numeric Strings) to Numeric Objects using Constructor of the corresponding Wrapper Class

Example 14.6

```
1.      class StringToNumericObjectDemo1
2.      {
3.          public static void main(String args[])
4.          {
5.              String str=new String("30");
6.              //String str="30";
7.              String str2=new String("30.333");
8.              Byte bObj = new Byte(str);
9.              System.out.println(bObj);
10.         //Byte bObj=new Byte(str2);          //NumberFormatException
11.         Short sObj=new Short(str);
12.         System.out.println(sObj);
13.         Integer iObj=new Integer(str);
14.         System.out.println(iObj);
15.         Long lObj=new Long(str);
16.         System.out.println(lObj);
17.         Float fObj=new Float(str);
18.         System.out.println(fObj);
19.         Double dObj=new Double(str);
20.         System.out.println(dObj);
21.     }
22. }
```

Output:

30

30

30

30

30.333

30.333

Note:- The Above constructor throw “NumberFormatException” if the string does not contain a parsable number.

Converting String Objects (Numeric Strings) to Primitive Numbers using parsing methods of the corresponding Wrapper Class

Example 14.7:

```
1.      class StringToPrimitiveDemo
2.      {
3.          public static void main(String args[])
4.          {
5.              String str = new String("30");
6.              //String str="30";
7.              String str2= new String("30.333");
8.              byte b = Byte.parseByte(str);
9.              System.out.println(b);
10.         //byte b1=Byte.parseByte(str2);           //NumberFormatException
11.         short s = Short.parseShort(str);
12.         System.out.println(s);
13.         int i = Integer.parseInt(str);
14.         System.out.println(i);
15.         long l = Long.parseLong(str);
16.         System.out.println(l);
17.         float f = Float.parseFloat(str2);
18.         System.out.println(f);
19.         double d = Double.parseDouble(str2);
20.         System.out.println(d);
21.     }
22. }
```

Output:

```
30
30
30
30
```

30.333

30.333

Note:-parseXXXXX() methods throw “NumberFormatException” if the string does not contain a parsable number.

Constants defined in classes Double and Float:

MAX_VALUE

MIN_VALUE

NaN

NEGATIVE_INFINITY

POSITIVE_INFINITY

TYPE (The class instance representing the primitive type double/float)

SIZE (The number of bits used to represent a double value). Since J2SDK 1.5.

Other Methods in Float Class:

1. **static int compare(float f1, float f2)**
2. **Compares the two specified float values**
3. **int compareTo(Float anotherFloat)**
4. **Compares two Float objects numerically.**
5. **boolean equals(Object obj)**
6. **Compares this object against the specified object**
7. **int hashCode()**
8. **Returns a hash code for this float object.**
9. **boolean isInfinite()**
10. **Returns true if this Float value is infinitely large in magnitude, false otherwise.**
11. **static boolean isInfinite(float v)**
12. **Returns true if the specified number is infinitely large in magnitude, false otherwise.**
13. **boolean isNaN()**
14. **Returns true if this float value is a Not-a-Number (NaN) value, false otherwise.**
15. **static Boolean isNaN(float v)**
16. **Returns true if the specified number is a Not-a-Number (NaN) value, false otherwise.**

Other methods in double Class:

Methods in Double class are almost same as methods of float Class, with use of double and Double words instead of float and Float words in the previous table.

Constants defined in classes Byte, Short, Integer and Long:

MIN_VALUE

MAX_VALUE

TYPE (The class instance representing the primitive type byte/short/int/long)

SIZE (The number of bits used to represent a byte/short/int/long value in two's complement binary form.) Since J2SDK 1.5.

Other Methods in Byte Class:

int compareTo(Byte anotherByte)

Compares two Byte objects numerically.

static Byte decode(String nm)

Returns a Byte object that contains the value specified by the string nm. Accepts decimal, hexadecimal, and octal numbers given by the following grammar:

DecodableString:

[-] DecimalNumerical

[-] 0x HexDigits

[-] 0X HexDigits

[-] # HexDigits

[-] 0 OctalDigits

boolean equals(Object obj)

Compares this object to the specified object.

int hashCode()

Returns a hash code for this Byte.

Other Methods in Short Class:

Methods in Short class are same as methods of Byte class, with the use of short and Short words instead of byte and Byte words in the above table.

Other Methods in Integer Class:

Some methods are same as Byte or Short class's methods. Some other methods are described below:

static String toBinaryString(int i)

Returns a string representation of the integer argument as an unsigned integer in base 2.

static String toHexString(int i)

Returns a string representation of the integer argument as an unsigned integer in base 16.

static String toOctalString(int i)

Returns a string representation of the integer argument as an unsigned integer in base 8.

Other Methods in Long Class:

Methods in Long class are almost same as methods of Integer Class.

Character Class:

The character class wraps a value of the primitive type char in an object.

Constructor:

Character(char value)

Constructs a newly allocated Character object that represents the specified char value.

Example: Character chObj = new Character('A');

Methods:

Some of the methods of Character class are described in the following table:

char charValue()

Returns the value of this Character object.

static boolean isDefined(char ch)

Determines if the specified character in Unicode.

static boolean isDigit(char ch)

Determines if the specified character is a digit.

static boolean isJavaIdentifierPart(char ch)

Determines if the specified character may be part of a Java identifier as other than the first character.

static boolean isJavaIdentifierStart(char ch)

Determines if the specified character is permissible as the first character in a Java

identifier.

static boolean isLetter(char ch)

Determines if the specified character is a letter.

static boolean isLetterorDigit(char ch)

Determines if the specified character is a letter or Digit.

static boolean isLowerCase(char ch)

Determines if the specified character is a lowercase character.

static boolean isSpaceChar(char ch)

Determines if the specified character is a Unicode space character.

static boolean isTitleCase(char ch)

Determines if the specified character is a titlecase character.

static boolean isUpperCase(char ch)

Determines if the specified character is a uppercase character.

static boolean isWhitespace(char ch)

Determines if the specified character is a whitespace according to Java.

static char toLowerCase(char ch)

Converts the character argument to lowercase.

static char toUpperCase(char ch)

Converts the character argument to uppercase.

static char toTitleCase(char ch)

Converts the character argument to Titlecase.

int compareTo(Character anotherCharacter)

Compares two Character objects numerically.

int hashCode()

Returns a hash code for this Character.

boolean equals(Object obj)

Compares this object against the specified object.

Boolean Class: The Boolean class wraps a value of the primitive type Boolean in an object.

Constructors:

Boolean(Boolean value)

Allocates a Boolean object representing the value argument.

Boolean(String s)

Allocates a Boolean object representing the value true if the string argument is not null and is equal, ignoring case, to the string “true”.

Methods of Boolean Class:

boolean booleanValue()

Returns the value of this Boolean object as a Boolean primitive.

int compareTo(Boolean b)

Compares this Boolean instance with another.

boolean equals(Object obj)

Returns true if and only if the argument is not a null and is a Boolean object that represents the same boolean value as this object.

static Boolean getBoolean(String name)

Returns true if and only if the system property named by the argument exists and is equal to the string “true”.

int hashCode()

Returns a hash code for this Boolean object.

static boolean parseBoolean(String s)

Parses the string argument as a boolean.

String toString()

Returns a String object representing this Boolean’s value.

static String toString(boolean b)

Returns a String object representing the specified boolean.

static Boolean valueOf(boolean b)

Returns a Boolean instance representing the specified boolean value.

static Boolean valueOf(String s)

Returns a Boolean with a value represented the specified String.



CHAPTER

∞ 15 ∞

(Input / Output in Java)

Introduction-

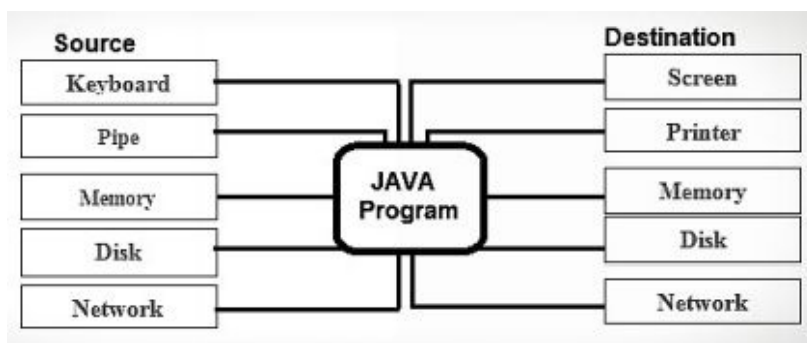
Streams-

Java programs perform I/O through streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system.

All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device.

This means an input stream can abstract many different kinds of input: from a disk file, a keyboard or a network socket. Likewise an output stream may refer to the console, a disk file, or a network connection.

Streams are a clean way to deal with input/output without having every part of our code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the **java.io** package.



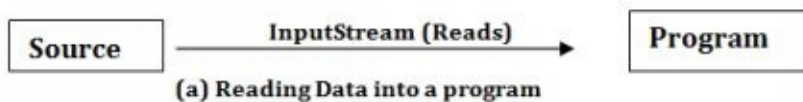
(Relationship of Java Program with I/O Devices)

The concept of sending data from one stream to another (like one pipe feeding into another pipe) has made streams in Java a powerful tool for file processing. We can build a complex file processing sequence using a series of simple stream operations.

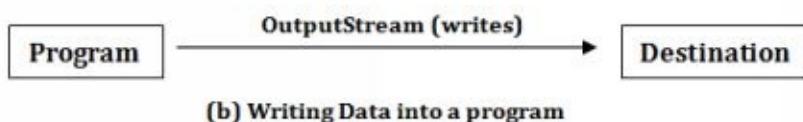
This feature can be used to filter data along the pipeline of streams so that we obtain data in a desired format. For example, we can use one stream to get raw data in binary format and then use another stream in series to convert it into integers.

Input and Output Streams:-

Java streams are classified into two basic types: normally input streams and output streams. An input stream extracts (i.e. reads) data from the source and sends it to the program.



Similarly, an output stream takes data from the program and sends (i.e. writes) it to the destination.



The program connects and opens an input stream on the data source and then reads the data serially. Similarly, the program connects and opens an output stream to the destination place of data and writes data out serially. In both the cases, the program does not know the details of end points (i.e. source and destination).

Byte Streams and Character Streams-

Java2 defines two types of streams: **Byte** and **Character**. Byte streams provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. Character streams provide a convenient means for handling input and output of character.

They use Unicode and therefore, can be internationalized. Also in some cases, character streams are more efficient than byte streams. The original version of Java (Java 1.0) did not include character streams and this, all I/O was byte oriented.

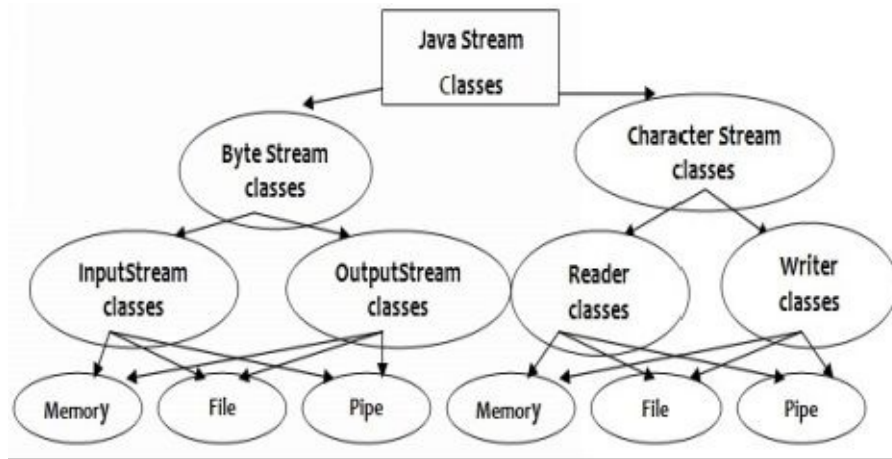
Character streams were added by Java 1.1 and certain byte oriented classes and methods were deprecated.

Note: At the lowest level, all I/O is still byte oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

These two groups may be further classified base on their purposes. **Byte Stream** and **Character Stream** classes contain specialized classes to deal with input and output operations independently on various types of devices.

We can also cross-group the streams based on the type of source or destination they read from or write to. The source (or destination may be memory, a file or a pipe.





(Classification of Java Stream Classes)

Overview of Byte Stream Classes:-

Byte stream classes are defined by using two class hierarchies. At the top are two abstract classes:

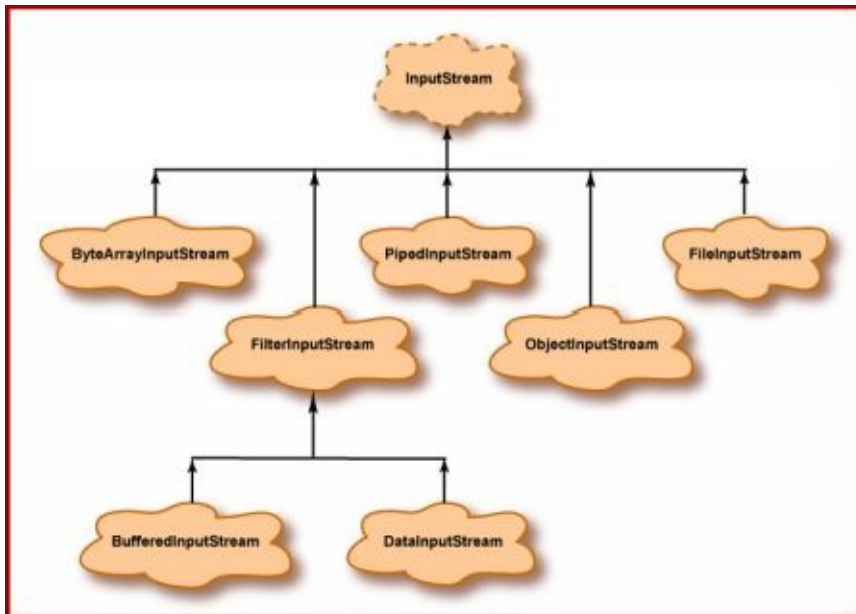
(1) `InputStream` and (2) `OutputStream`

Each of these abstract classes have several concrete subclasses, that handle the differences between various devices, such as disk files, network connections and even memory buffers.

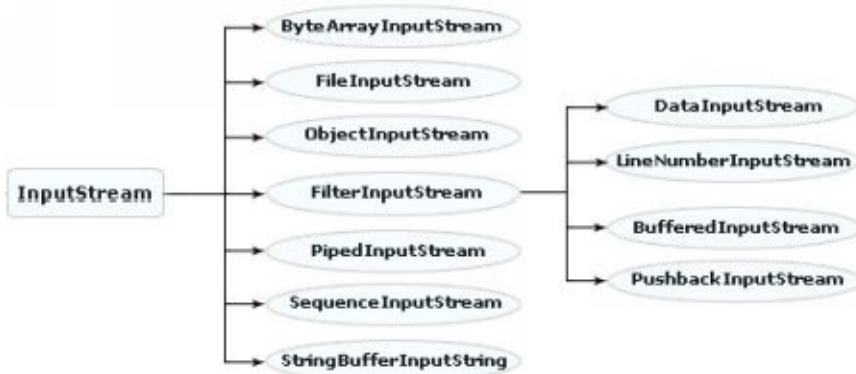
`InputStream` classes

`InputStream` classes that are used to read 8-bit bytes include a super class known as `InputStream` and a number of sub-classes for supporting various input related functions.





Hierarchy of Input Stream Classes.



Hierarchy of Input Stream Classes.

The InputStream class:-

The super class InputStream is an abstract class, which defines methods for performing input functions such as:

- ✚ Reading bytes.
- ✚ Closing stream.
- ✚ Marking positions in streams.
- ✚ Skipping ahead in a stream.
- ✚ Finding the number of bytes in a stream.

Methods of InputStream class:

int available()

Gives the number of bytes available in the input (must be overridden by the subclasses). The available method for class InputStream always returns 0.

void close()

Closes this input stream and releases any system resources associated with the stream. The close method of InputStream does nothing.

void mark(int readlimit)

Marks the current position in this input stream. The mark method of InputStream does nothing.

boolean markSupported()

Tests if this input stream supports the mark and reset methods. The markSupported method of InputStream return false.

abstract int read()

Reads the next byte of data from the input stream. A subclass must provide an implementation of this method.

int read(byte[] b)

Reads some number of bytes from the input stream and stores them into the buffer array b. The number of bytes actually read is returned as an integer.

int read(byte[] b, int offset, int len)

Reads some number of bytes from the input stream and stores them into the buffer array b starting from position specified by the parameter off. The number of bytes actually read is returned as an integer.

void reset()

Repositions this stream to the position at the time the mark method was

last called on this input stream. The reset method of InputStream does nothing and always throws IOException.

long skip(long n)

Skips over and discards upto n bytes of data from this input stream and returns the number of bytes actually skipped.

The DataInput Interface:

The DataInput is an interface, which defines methods for reading primitives from an InputStream.

Methods of DataInput Interface:

boolean readBoolean()

Reads one input byte and returns true if that byte is non-zero and false if that byte is zero.

byte readByte()

Reads and returns one byte input.

char readChar()

Reads an input character and returns its value.

double readDouble()

Reads eight input bytes and returns a double value.

float readFloat()

Reads four input bytes and returns a double value.

void readFully(byte[] b)

Reads some bytes from an input stream and stores them in a buffer array **b**. This method throws EOFException if this stream reaches the end before reading all the bytes.

void readFully(byte[] b, int off, int len)

Reads **len** bytes from an input stream and stores them in a buffer array **b** starting at

the position specified by the **off**. This method throws EOFException if this stream reaches the end before reading all the bytes.

int readInt()

Reads four input bytes and returns an int value.

String readLine()

Reads the next line from the input stream.

long readLong()

Reads eight input bytes and returns a long value.

short readShort()

Reads two input bytes and returns a short value

String readUTF()

Reads in a string that has been encoded using a modified UTF-8 format.

int skipBytes(int n)

Makes an attempt to skip over n bytes of data from the input stream, discarding the skipped bytes. The actual number of bytes skipped is returned.

Summary of InputStream Classes:-

A FilterInputStream contains some other input stream, which it uses as its basic source of data possibly transforming the data along the way providing additional functionality.

The class FilterInputStream itself simply overrides all methods of InputStream with versions that pass all requests to the contained input stream. Sub-classes of FilterInputStream may further override some of these methods and may also provide additional methods and fields.

Note that the class DataInputStream extends FilterInputStream and implements

the interface `DataInput`. Therefore, the `DataInputStream` extends `FilterInputStream` and implements the methods described in `DataInput` in addition to using methods of `InputStream` class.

InputStream classes summary:

InputStream	This abstract class is the superclass of all classes representing an input stream of bytes.
BufferedInputStream	A <code>bufferedInputStream</code> adds functionality to another input stream-namely, the ability to buffer the input and to support the mark and reset method.
ByteArrayInputStream	A <code>ByteArrayInputStream</code> contains an internal buffer that contain bytes that may be read from the stream.
DataInputStream	A <code>DataInputStream</code> lets an application read the primitive Java data types from an underlying input stream from a machine-independent way.
FileInputStream	A <code>FileInputStream</code> contains input bytes from a file in a file system.
FilterInputStream	A <code>FilterInputStream</code> contains some other input stream, which it uses as its basic source of data, possibly transferring the data along the way or providing additional functionality.
PipedInputStream	A <code>PipedInputStream</code> should be connected to a piped output stream, the piped input stream then provides whatever data bytes are written to piped output stream.
PushbackInputStream	A <code>PushbackInputStream</code> adds functionality to another input stream, namely the ability to “pushback” or “unread” one byte.
SequenceInputStream	A <code>SequenceInputStream</code> represents the logical concentration of another stream.
ObjectInputStream	An <code>ObjectInputStream</code> deserializes primitive data and objects previously written using an <code>ObjectOutputStream</code> .

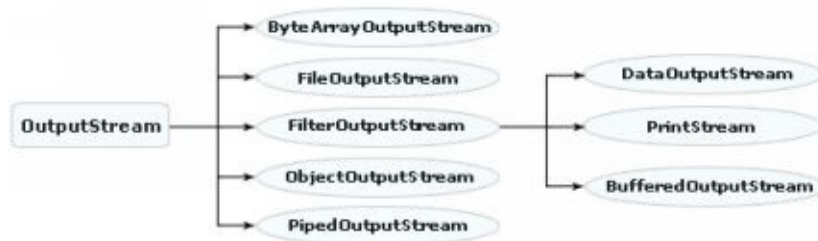
OutputStream classes-

Output stream classes are derived from the base class `OutputStream`, which is an abstract class and have a number of sub-classes for supporting various output related functions.

The OutputStream class:-

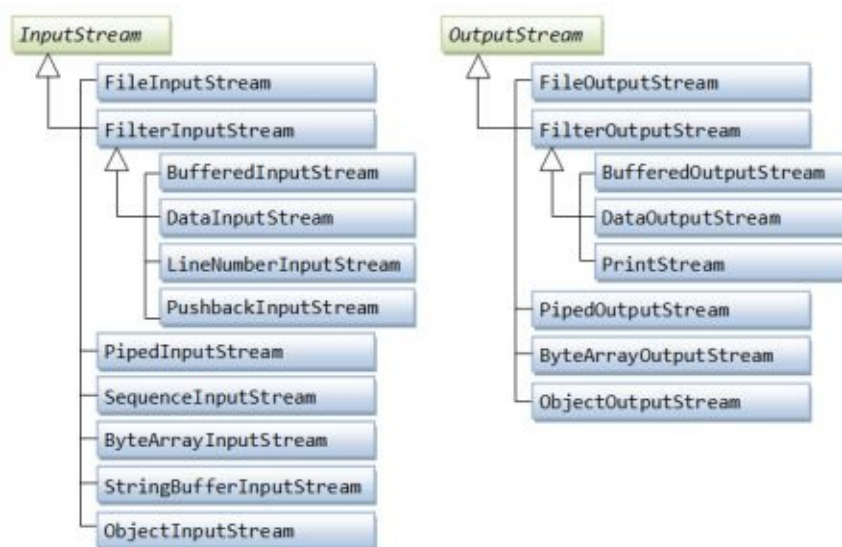
The super class OutputStream is an abstract class, which defines methods for performing output functions such as:

- ✚ Writing bytes
- ✚ Closing streams
- ✚ Flushing streams



(Hierarchy of Output Stream Classes)

Oops !! Let's see following good quality picture-



Methods of OutputStream Class:-

void close()

Closes this output stream and releases any stream resources associated with this stream.

void flush()

Flushes this output stream and forces any buffered output bytes to be written out.

void write(byte[] b)

Writes b.length bytes from the specified byte array to its output stream.

void write(byte[] b, int off, int len)

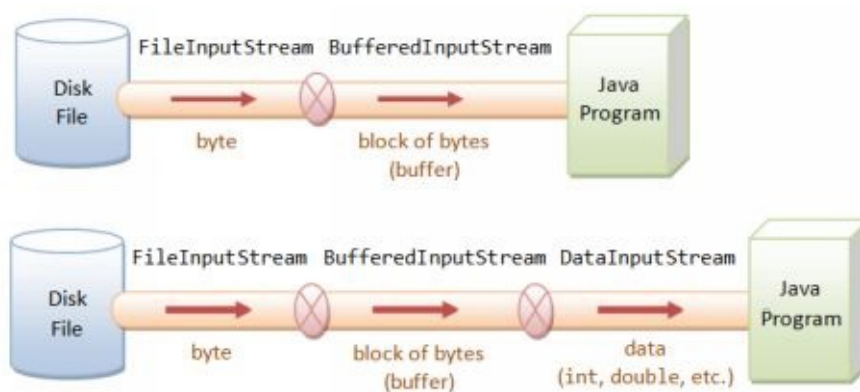
Writes len bytes from the specified byte array starting at offset off to its output stream.

abstract void write(int b)

Writes the specified byte to its output stream.

The DataOutput Interface:-

The DataOutput is an interface, which defines methods for writing primitives to an OutputStream.



Methods of DataOutput Interface:

void writeBoolean(boolean v)

Writes a boolean value to its output stream.

void writeByte(int v)

Writes to the output stream the eight low- order bits of the argument v.

void writeBytes(string s)

Writes the string to the output stream. For every character in the string s, taken in order, one byte is written to the output stream.

void writeChar(int v)

Writes a character value, which is comprised of two bytes, to the output stream.

void writeChars(string s)

Writes every character in the string s, to the output stream in the order of eight bytes, to the output stream.

void writeDouble(double v)

Writes a double value, which is comprised of eight bytes, to the output stream.

void writeFloat(float v)

Writes a float value, which is comprised of four bytes, to the output stream.

void writeInt(int v)

Writes a int value, which is comprised of four bytes, to the output stream.

void writeLong(long v)

Writes a long value, which is comprised of eight bytes, to the output stream.

void writeShort(int v)

Writes a short value, which is comprised of two bytes, to the output stream.

void writeUTF(String str)

Writes two bytes of length information to the output stream, followed by the modified UTF-8 representation of every character in the string s.

Summary of OutputStream Classes:-

The `DataOutputStream` is a counter part of `DataInputStream`. The `DataOutputStream` class implements the methods described in `DataOutput` interface in addition to using methods of `OutputStream` class.

OutputStream Classes Summary:-

OutputStream	This abstract class is the super class of all classes representing an output stream of bytes.
BufferedOutputStream	The class implements a buffered output stream.
ByteArrayOutputStream	The class implements an output stream in which the data is written into a byte array.
DataOutputStream	A data output stream lets an application write primitive Java data types to an output stream in a portable way.
FileOutputStream	A file output stream is an output stream for writing data to a File.
PipedOutputStream	A piped output stream can be connected to a piped input stream to create a communications pipe.
FilterOutputStream	A FilterOutputStream contains some other output streams, which it uses as its destination, possibly transforming the data along the way or providing the additional functionality.
PrintStream	A PrintStream adds functionality to another output stream, namely the ability to print representations of various data values conveniently.

Overview of Character Stream Classes-

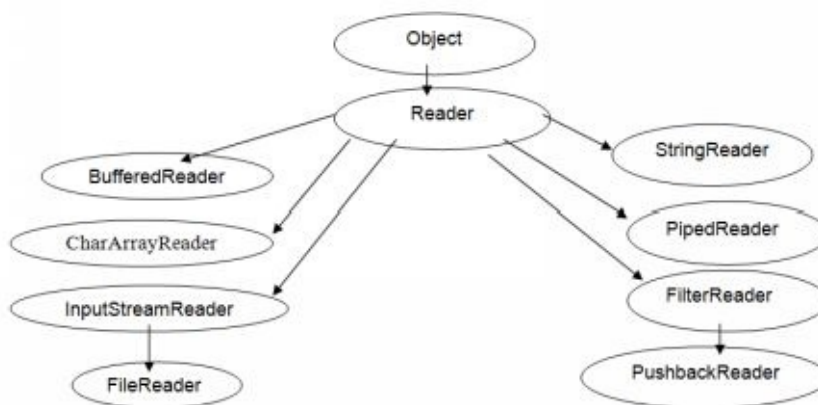
Character streams are defined by using two class hierarchies. At the top are two abstract classes: Reader and Writer. These abstract classes handle Unicode character streams.

The Java has several concrete subclasses of each of these. The abstract Reader and Writer classes define several key methods that the other stream classes implement. Two of the most important methods are read() and write(), which read and write characters of data, respectively. These methods are overridden by derived stream classes.

Reader stream classes:-

Reader stream classes are designed to read character form the files. Reader class is the base class for all other classes in this group. These classes are functionally very similar to the input stream classes, except input streams use bytes as their fundamental unit of information, while reader streams use characters.

The Reader class contains methods that are identical to those available in the InputStream class. Therefore, Reader classes can perform almost all the functions implemented by the input stream classes.



(Hierarchy of Reader stream Class)

The Reader Class:-

Reader is an abstract class that defines Java’s model of streaming character input. All of the methods in this class will throw an IOException.

Methods of Reader Class:-

abstract void close()

Closes the stream. Once a stream has been closed, further read(), ready(), mark(), or reset() invocations will throw an IOException. Closing a previously closed

stream, however, has no effect.

void mark()

Marks the present position in the stream. Subsequently calls to `reset()` will attempt to reposition the stream to this point. Not all character input streams support the `mark()` operation.

boolean markSupported()

Tells whether this stream supports the `mark()` operation

int read()

Reads a single character and returns the integer representation of the next available character from the invoking input stream. -1 is returned when end of file is encountered.

int read(char[] buffer)

Attempts to read up to buffer length characters into buffer and returns the actual number of characters that are successfully read. -1 is returned when end of file is encountered. The subclasses of the reader must implement this.

abstract int read(char[] buffer, int offset, int numChar)

Attempts to read up to numChar characters starting at `buffer[offset]`, returning the number of characters that are successfully read. -1 is returned when end of file is encountered. The subclasses of the reader must implement this.

boolean ready()

Tells whether this stream is ready to be read. Returns true if the next input request will not wait, otherwise it returns a false.

void reset()

Reset the stream. This method is not supported by all character input streams.

long skip(long n)

Skips up to n characters and returns the number of characters skipped. This throws `IllegalArgumentException` if n is -ve.

Summary of Reader Classes:-

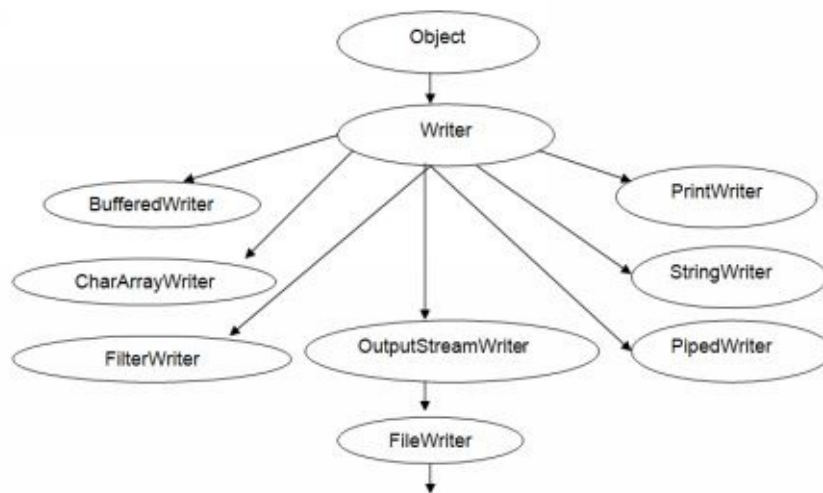
Reader	Abstract class for reading character streams
BufferedReader	Read text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays and lines.
CharArrayReader	This class implements a character buffer that can be used as a character-input stream.
FileReader	Convenience class for reading character files.
FilterReader	Abstract class for reading filter character streams.
InputStreamReader	An InputStreamReader is a bridge from byte streams to character stream: It reads bytes and decodes them into characters using a specified charset.
PipedReader	Piped character input stream.
PushbackReader	A character stream reader that allows characters to be pushed back into the stream.
StringReader	A character stream whose source is a string.

Writer Stream Class:-

Writer Stream Classes are designed to write character to files/output devices. The writer class is an abstract class, which acts as a base class for all the other writer stream class.

The base class provides support for all output operations by defining methods that are identical in those in OutputStream class.

The Writer class contains methods that are identical to those available in the OutputStream class. Therefore, Writer classes can perform almost all the functions implemented by the output stream classes.



(Hierarchy of Writer Stream Classes)



The Writer Class:-

Writer is an abstract class that defines streaming character output. All of the methods in this class return a void value and throw an IOException in the case of errors.

Methods of Writer class:

abstract void close()

Closes the stream flushes it first.

abstract void flush()

Flushes the stream.

void write(char[] cbuf)

Writes an array of characters.

abstract void write(char[] cbuf, int off, int len)

Writes a portion of an array of characters beginning from specified offset. The subclasses must implement this method.

void write(int c)

Writes a single character to the invoking output stream. Note that the parameter is an int, which allows you to call write with expressions without having to cast them back to char.

void write(String str)

Writes a string.

void write(String str, int off, int len)

Writes the portion of the string beginning at the specified offset.



Summary of Writer Class-

Writer	Abstract class for writing to character streams.
BufferedWriter	Writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.
CharArrayWriter	This class implements a character buffer that can be used as a Writer.
FileWriter	Convenience class for writing character files.
FilterWriter	Abstract class for writing filtered character streams.
OutputStreamWriter	An OutputStreamWriter is a bridge from character streams to byte streams: Characters written to it are encoded into bytes using a specified charset.
PipedWriter	Piped character-output streams.
PrintWriter	Print formatted representations of objects to a text-output stream.
StringWriter	A character stream that collects its output in a string buffer, which can then be used to construct a string.

File Class-

Although most if the classes defined by java.io operate on streams, the File class

does not. It deals directly with files and the file system. That is, the File class does not specify how information is retrieved from or stored in files. It describes the properties of a file itself.

A File object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date and directory path, and to navigate sub-directory hierarchies.

Although there are severe restrictions on their use within applets for security reasons, files are still a central resource for storing persistent and shared information.

A directory in Java is treated simply as a file with one additional property – a list of filenames that can be examined by the list() method.

```
public File(String);
```

```
public File(String, String);
```

```
public File(File, String);
```

```
public File(URI);
```

This constructor was added by Java2, version 1.4

```
public String getName();
```

Returns the name of the file or directory

```
public String getParent();
```

Returns the parent directory of the file from the path specified during object creation, or null if this pathname does not name a parent directory.

```
public String getPath();
```

Returns the relative path.

public boolean isAbsolute();

Tests whether this abstract pathname is absolute.

public String getAbsolutePath();

Returns the absolute(complete) path starting from root.

public boolean canRead();

Tests whether the application can read the file denoted by this abstract pathname.

public boolean canWrite();

Tests whether the application can modify the file denoted by this abstract pathname.

public boolean exists();

Tests whether the file or directory denoted by this abstract pathname exists.

public boolean isDirectory();

Tests whether the file denoted by this abstract pathname is a directory.

public boolean isFile();

Tests whether the file denoted by this abstract pathname is normal file.

public boolean isHidden();

Tests whether the file named by this abstract pathname is a hidden file.

public long lastModified();

Returns the time that the file denoted by this abstract pathname was last modified.

public long length();

Returns the length of the file denoted by this abstract pathname.

public boolean createNewFile() throws java.io.IOException;

Automatically creates a new, empty file named by this abstract pathname if and only if a file with this name does not yet exist.

public boolean delete();

Deletes the file or directory denoted by this abstract pathname.

public String[] list();

Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname. Returns null if File object corresponds to a file.

public boolean mkdir();

Creates the directory named by this abstract pathname.

public boolean mkdirs();

Creates the directory named by this abstract pathname including any necessary but nonexistent parent directories.

public boolean setLastModified(long);

Sets the last modified time of the file or directory named by this abstract pathname.

public boolean setReadOnly();

Marks the file or directory named by this abstract pathname so that only readoperations are allowed.

public boolean setWritable(boolean);

```
public boolean setReadable(boolean);
```

```
public boolean canExecute( );
```

```
public long getTotalSpace( );
```

```
public long getFreeSpace( );
```

```
public long getUsableSpace( );
```

```
boolean renameTo(File);
```

Rename the file denoted by this abstract pathname.

Examples-

```
File f1 = new File("/");
```

```
File f2 = new File("/", "autoexec.bat");
```

```
File f3 = new File(f1, "autoexec.bat");
```

Note:

Java does the right thing With path separators between UNIX and Windows conventions. If you use a forward slash (/) on a windows version of java, the path will still resolve correctly.

Remember, if you are using the windows convention (\) with a string. The Java convention is to use the UNIX – and URL style forward slash for path separators.

File defines many methods that obtain the standard properties of a File object. The File class however, is not symmetrical. By this, we mean that there are main methods that allow you to examine the properties of a simple file object, but no corresponding function exists to change these attributes.

Example 15.1:

Following example demonstrates several methods of the class File

```
1.      import java.io.*;
```

```

2.     class FileTest
3.     {
4.         public static void main(String args[]) throws IOException
5.         {
6.     File f = new File("/javaprogram/ch15/15.1/FileTest.java");
7.         if(!f.exists())
8.             f.createNewFile();
9.         System.out.println("Length="+f.length()+" bytes");
10.        System.out.println("Name="+f.getName());
11.        System.out.println("Parent="+f.getParent());
12.        System.out.println("Path="+f.getPath());
13.        System.out.println("Absolute Path="+f.getAbsolutePath());
14.        System.out.println(f.exists() ? "Exists" : "Does not exist");
15.        System.out.println(f.isFile() ? "is file" : "not a file");
16.        System.out.println(f.canRead() ? "is readable" : "not readable");
17.        System.out.println(f.canWrite() ? "is writable" : "not writable");
18.        System.out.println(f.isDirectory() ? "is directory" : "not a directory");
19.        System.out.println(f.isHidden() ? "is hidden" : "not hidden");
20.        System.out.println(f.isAbsolute() ? "is absolute" : "is not absolute");
21.        System.out.println("File last modified :"+f.lastModified());
22.        //f.delete();
23.        File f3=new File("C:/tally72");
24.        String nm[ ] = f3.list();
25.        System.out.println("List of Files & Sub-Directories of c:/tally72");
26.        for(int i=0;i<nm.length;i++)
27.            {
28.                System.out.print(nm[i]);
29.                File f4 = new File("c:/tally72/"+nm[i]);
30.                if(f4.isFile() == true)
31.                    System.out.println(" It is a File");
32.                else
33.                    System.out.println(" It is a Directory");

```



```
34.         }
35.         File f5=new File("zzz");
36.         f5.mkdir();
37.         File f6=new File("yyy/aaa");
38.         f6.mkdirs();
39.     }
40. }
```

Output:

Length=1782 bytes

Name=FileTest.java

Parent=\javaprogram\ch15\15.1

Path=\javaprogram\ch15\15.1\FileTest.java

Absolute Path=C:\javaprogram\ch15\15.1\FileTest.java

Exists

is file

is readable

is writable

not a directory

not hidden

is not absolute

File last modified :1236824168000

List of Files & Sub-Directories of c:/tally72

TALLY.REW It is a File

Tally.ini It is a File

sentinel.sys It is a File

sentinel.vxd It is a File

spnsrv9x.exe It is a File

spnsrvnt.exe It is a File

stat.slk It is a File

tally72.exe It is a File

tallylic9xserver.exe It is a File

tallylicserver.exe It is a File

tallysav.dat It is a File

tallywin.dat It is a File

tally.imp It is a File

Data It is a Directory

FileInputStream-

It is a direct sub-class of the Input Stream class. To open a file for reading, you simply create an object of FileInputStream class, specifying the name of the file (directly or indirectly) as an argument to the constructor.

```
public FileInputStream(String) throws FileNotFoundException;
```

Creates a FileInputStream by opening a connection to an actual file, the file named by the path name filePath in the file system.

```
public FileInputStream(File) throws FileNotFoundException;
```

Creates a FileInputStream by opening a connection to an actual file, the file named by the File object file in the file system.

```
public native int read( ) throws IOException;
```

```
public int read(byte[ ]) throws IOException;
```

```
public int read(byte[ ], int, int) throws IOException;
```

```
public native long skip(long) throws IOException;
```

```
public native int available( ) throws IOException;
```

```
public void close( ) throws IOException;
```

FileOutputStream-

It is direct sub-class of the OutputStream class. To open a file for writing, we simply create an object of FileOutputStream class, specifying the name of the file (directly or indirectly) as an argument to the constructor.

public FileOutputStream(String) throws FileNotFoundException;

Creates an output file stream to write to the file with the specified name.

public FileOutputStream(String, boolean)

throws java.io.FileNotFoundException;

Create an output file stream to write to the file with the specified name.

public FileOutputStream(File) throws FileNotFoundException;

Creates a file output stream to write to the file represented by the specified File object.

public FileOutputStream(File, boolean) throws FileNotFoundException;

Creates a file output stream to write to the file represented by the specified File object.

public native void write(int) throws java.io.IOException;

public void write(byte[]) throws java.io.IOException;

public void write(byte[] , int, int) throws java.io.IOException;

public void close() throws java.io.IOException;

Example 15.2

```
1.      import java.io.*;
2.      class FileRead
3.      {
4.          public static void main(String args[])
5.          {
6.              FileInputStream fin=null;
7.              int a;
8.              try
9.              {
10.         fin=new FileInputStream(args[0]);
11.         System.out.println("No. of characters to read="+fin.available());
12.         fin.skip(3);
```

```
13.             a = fin.read();
14.             while(a != -1)
15.             {
16.                 System.out.print((char)a);
17.                 a = fin.read();
18.             }
19.     System.out.println("No. of characters to read="+fin.available());
20.             fin.close();
21.     }
22.     catch(ArrayIndexOutOfBoundsException e)
23.     {
24.         System.out.println(e);
25.     }
26.     catch(FileNotFoundException e)
27.     {
28.         System.out.println(e);
29.     }
30.     catch(IOException e)
31.     {
32.         System.out.println(e);
33.     }
34.     }
35. }
```

Output:

```
java FileRead A.txt
```

```
No. of characters to read=20
```

```
Matrix Computers
```

```
No. of characters to read=0
```

Example 15.3:

```
1.      import java.io.*;
2.      class FileCopy
3.      {
4.          public static void main(String args[]) throws IOException
5.          {
6.              FileInputStream fin=null;
7.              FileOutputStream fout=null;
8.              fin=new FileInputStream(args[0]);
9.              fout=new FileOutputStream(args[1]);
10.             int a;
11.             a = fin.read();
12.             while(a != -1)
13.             {
14.                 fout.write(a);
15.                 a = fin.read();
16.             }
17.             fin.close();fout.close();
18.         }
19.     }
```

Example 15.4

```
1.      import java.io.*;
2.      class CopyFile
3.      {
4.          public static void main(String args[]) throws IOException
5.          {
6.              int n;
7.              FileInputStream fin = new FileInputStream("A.txt");
```

```

8.          FileOutputStream fout = new FileOutputStream("B.txt");
9.          byte b[ ]= new byte[10];
10.         n = fin.read(b);
11.         while(n != -1)
12.         {
13.             fout.write(b,0,n);
14.             n = fin.read(b);
15.         }
16.         fin.close();
17.         fout.close();
18.     }
19. }

```

FileReader-

The FileReader class creates a Reader that you can use to read the contents of a file. Its two most commonly used constructors are:

FileReader(File file)-

Creates a FileReader stream to read from the specified file.

FileReader(String fileName) -

Creates a FileReader stream to read from the specified file.

Note: Either of the constructors can throw a FileNotFoundException if the file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

FileWriter

The FileWriter class creates a writer that you can use to write to a file. Its two most commonly used constructors are:

Constructors:-

FileWriter(File file)

Constructors a FileWriter object to write to the specified file.

FileWriter(String fileName)

Constructors a FileWriter object to write to the specified file.

FileWriter(File file, boolean append)

Constructors a FileWriter object to write to the specified file in the append mode.

FileWriter(String filename, boolean append)

Constructors a FileWriter object to write to the specified file in the append mode. If first two constructors are used then always a new file is created. If file with the specified name exists, it gets destroyed. If the last two constructors are used and append flag is false then they behave like first two constructors.

If the append flag is true then file is opened in the append mode i.e. if file already exists then file pointer is placed at the end of the file and if file does not exist then new file is created.

Note:

These constructors can throw an IOException if the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason.

For example, when you attempt to open a read-only file, an IOException will be thrown.

Example 15.5

```
1.      import java.io.*;
2.      class FileCopy
3.      {
4.          public static void main(String args[]) throws IOException
5.          {
6.              FileReader fin=null;
7.              FileWriter fout=null;
8.              fin=new FileReader(args[0]);
9.              fout=new FileWriter(args[1]);
```

```

10.         int a;
11.         a = fin.read();
12.         while(a != -1)
13.         {
14.             fout.write((char)a);
15.             a = fin.read();
16.         }
17.         fin.close();
18.         fout.close();
19.     }
20. }

```

RandomAccessFile-

RandomAccessFile encapsulates a random-access file. The RandomAccessFile enables us to read and write bytes, text and primitive data types from/to location in a file (when used with appropriate access permissions). It is not derived from InputStream. Instead, it implements the interfaces DataInput and DataOutput, which define the basic I/O methods. It also supports positioning requests- that is, you can position the file pointer anywhere within the file

The mode argument specifies the access mode in which the file is to be opened. The permitted values and their meanings are:

Value	Meaning
--------------	----------------

“r”	Open for reading only. Invoking any of the write methods of the resulting object will cause an IOException to be thrown.
------------	--

“rw”	Open for reading and writing. If the file does not already exist then an attempt will be made to create it.
-------------	---

“rws”	Open for reading and writing, as with “rw”, and also require that every update to the file’s content or metadata be written synchronously to the underlying storage device.
--------------	---

“rwd” Open for reading and writing, as with “rw”. And also require that every update to the file’s content (data) be written synchronously to the underlying storage device.

Methods of RandomAccessFile class:-

Beside methods of the DataInput and DataOutput interfaces, some of the commonly used methods of this class are:

public RandomAccessFile(String, String) throws java.io.FileNotFoundException;

Creates a random access file stream to read from, and optionally to write to, the file specified name.

public RandomAccessFile(File, String) throws FileNotFoundException;

Creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.

public native int read() throws java.io.IOException;

public int read(byte[], int, int) throws java.io.IOException;

public int read(byte[]) throws java.io.IOException;

public int skipBytes(int) throws java.io.IOException;

public native void write(int) throws java.io.IOException;

public void write(byte[]) throws java.io.IOException;

public void write(byte[], int, int) throws java.io.IOException;

public native long getFilePointer() throws java.io.IOException;

Returns the offset from the beginning of the file, in bytes, at which the next read or write occurs.

public native void seek(long) throws java.io.IOException;

Here, newPos specifies the new position, in bytes of the file pointer from the beginning of the file. After a call to seek(), the next read or write operation will occur at the new file position.

public native long length() throws java.io.IOException;

public native void setLength(long) throws java.io.IOException;

Sets the length of this file. This method can be used to lengthen or shorten a file. If the file is lengthened, the added portion is undefined.

public void close() throws java.io.IOException;

public final boolean readBoolean() throws java.io.IOException;

public final byte readByte() throws java.io.IOException;

public final int readUnsignedByte() throws java.io.IOException;

public final short readShort() throws java.io.IOException;

public final int readUnsignedShort() throws java.io.IOException;

public final char readChar() throws java.io.IOException;

public final int readInt() throws java.io.IOException;

public final long readLong() throws java.io.IOException;

public final float readFloat() throws java.io.IOException;

public final double readDouble() throws java.io.IOException;

public final java.lang.String readLine() throws java.io.IOException;

public final java.lang.String readUTF() throws java.io.IOException;

public final void writeBoolean(boolean) throws java.io.IOException;

public final void writeByte(int) throws java.io.IOException;

public final void writeShort(int) throws java.io.IOException;

public final void writeChar(int) throws java.io.IOException;

public final void writeInt(int) throws java.io.IOException;

public final void writeLong(long) throws java.io.IOException;

public final void writeFloat(float) throws java.io.IOException;

public final void writeDouble(double) throws java.io.IOException;


```

22.             file.seek(file.length());
23.             file.writeBoolean(false);
24.             file.seek(14);
25.             System.out.println(file.readBoolean());
26.         System.out.println("File pointer Pos:"+file.getFilePointer());
27.         System.out.println(file.readInt());//EOF Expection
28.             }
29.         catch(EOFException e)
30.         {
31.             System.out.println("Trying to read after end of file");
32.         }
33.         finally
34.         {
35.             file.close();
36.         }
37.
38.     }
39. }

```

Output:

```

File pointer Pos:14
X
1999
222.44
File pointer Pos:14
File pointer Pos:2
1999
false
File pointer Pos:15
Trying to read after end of file

```

Note: An EOFException occurs if attempt is made to read after end of file.

ByteArrayInputStream

A ByteArrayInputStream contains an internal buffer that contain bytes that may be read from the stream. ByteArrayInputStream is an implementation of an input stream that uses a byte array as the source.

This class has two constructors, each of which requires a byte array to provide the data source.

A ByteArrayInputStream implement both mark() and reset() methods. However, if mark() has not been called, then reset() sets the stream pointer to the star of the stream.

```
public ByteArrayInputStream(byte[]);  
  
public ByteArrayInputStream(byte[], int, int);  
  
public synchronized int read();  
  
public synchronized int read(byte[], int, int);  
  
public synchronized long skip(long);  
  
public synchronized int available();  
  
public boolean markSupported();  
  
public void mark(int);  
  
public synchronized void reset();  
  
public void close()    throws IOException;
```

Example 15.7:

```
1.     import java.io.*;
2.     class ByteArrayInputStreamTest
3.     {
4.         public static void main(String args[]) throws IOException
5.         {
6.             String s1="abcdefghijk";
7.             byte b[]=s1.getBytes();
8.             ByteArrayInputStream b1 = new ByteArrayInputStream(b);
9.             int a;
10.            while((a=b1.read())!=-1)
11.            {
12.                System.out.print((char)Character.toUpperCase(a));
13.            }
14.            b1.close();
15.        }
16.    }
```

Output: ABCDEFGHIJK

ByteArrayOutputStream-

The class implements an output stream in which the data is written into a byte array. `ByteArrayOutputStream` is an implementation of an output stream that uses a byte array as the destination.

The buffer is hold in the protected **buf** field of `ByteArrayOutputStream`. The buffer size will be increased automatically, if needed. The number if bytes hold by the buffer is contained in the protected count field of `ByteArrayOutputStream`.

```
public java.io.ByteArrayOutputStream();
```

Creates a new byte array output stream. The buffer capacity is initially 32 bytes, though its size increases if necessary.

```
public java.io.ByteArrayOutputStream(int);
```

Creates a new byte array output stream, with a buffer capacity of the specified size, in bytes.

```
public synchronized void write(int);
```

```
public synchronized void write(byte[], int, int);
```

```
public synchronized void writeTo(java.io.OutputStream) throws  
java.io.IOException;
```

```
public synchronized void reset();
```

```
public synchronized byte[] toByteArray();
```

```
public synchronized int size();
```

```
public synchronized java.lang.String toString();
```

```
public synchronized java.lang.String toString(java.lang.String) throws  
java.io.UnsupportedEncodingException;
```

```
public synchronized java.lang.String toString(int);
```

```
public void close() throws java.io.IOException;
```

Example15.8

```
1. import java.io.*;  
2. class ByteArrayOutputStreamTest  
3. {  
4.     public static void main(String args[]) throws IOException  
5.     {  
6.         ByteArrayOutputStream b1 = new ByteArrayOutputStream();  
7.         b1.write('a');  
8.         String s1 = "abcdefghijk";
```

```

9.         byte arr1[ ]= s1.getBytes();
10.        b1.write(arr1);
11.        System.out.println(b1); //toString()
12.        //Display using for loop
13.        byte arr2[ ] = b1.toByteArray();
14.        for(int i=0;i<arr2.length;i++)
15.        {
16.            System.out.print((char)arr2[i]);
17.        }
18.        System.out.println();
19.        //Display using for each loop
20.        for(int i:arr2)
21.        {
22.            System.out.print((char)i);
23.        }
24.        b1.close();
25.    }
26. }

```

Output:

```

aabcdefghijk
aabcdefghijk
aabcdefghijk

```

CharArrayReader-

The CharArrayReader is an implementation of a reader stream that uses a char array as the source. This class has two constructors, each of which requires a char array to provide the data source.

Constructors:

CharArrayReader(char[]buf)

Crates a CharArrayReader from the specified array of chars.

CharArrayReader(char[]buf, int offset, int length)

Crates a CharArrayReader from the specified array of chars. The resulting reader will start reading at the given offset. The total number of char values that can be read from this reader will be either length or buf.length.

Example 15.9

```
1.      import java.io.*;
2.      class CharArrayReaderTest
3.      {
4.          public static void main (String args[]) throws IOException
5.          {
6.              String tmp = "abcdefghijklmnopqrstuvwxyz";
7.              int length = tmp.length();
8.              char c[ ] = new char[length];
9.              tmp.getChars(0,length,c,0);
10.         CharArrayReader input1 = new CharArrayReader(c);
11.         CharArrayReader input2 = new CharArrayReader(c,0,5);
12.             int i;
13.             while((i=input1.read()) != -1)
14.             {
15.                 System.out.print((char)i);
16.             }
17.             System.out.println();
18.             while((i=input2.read()) != -1)
19.             {
20.                 System.out.print((char)i);
21.             }
22.         }
23.     }
```

Output:

abcdefghijklmnopqrstuvwxy

abcde

CharArrayWriter

CharArrayWriter is an implementation of a writer stream that uses a char array buffer as the destination.

Constructors:

CharArrayWriter()

CharArrayWriter(int initialSize)

In the first form, a buffer with a default size is created. In the second, a buffer is created with the size equal to that specified by the parameter initialSize.

The buffer is hold in the buf field of CharArrayWriter. The buffer size will be increased automatically, if needed. The number if character held by the buffer is contained in the count field of CharArrayWriter. Both buf and count are protected fields.

Example 15.10

```
1.      import java.io.*;
2.      class CharArrayWriterTest
3.      {
4.          public static void main (String args[]) throws IOException
5.          {
6.              CharArrayWriter f = new CharArrayWriter();
7.              String s = "This should end up in the array";
8.              char buf[] = new char[s.length()];
9.              s.getChars(0,s.length(),buf,0);
10.             f.write(buf);
11.             System.out.println(f.toString());
12.             char c[ ] = f.toCharArray();
13.             for(int i=0; i<c.length;i++)
```

```

14.         {
15.             System.out.print(c[i]);
16.         }
17.     FileWriter f2 = new FileWriter("test.txt");
18.     f.writeTo(f2);
19.     f2.close();
20.     f.reset();
21.     System.out.println();
22.     System.out.println("After reset:"+f.toString());
23.     for(int i=0;i<3;i++)
24.         f.write('A');
25.     System.out.println(f.toString());
26.     }
27. }

```

Output:

This should end up in the array

This should end up in the array

After reset:

AAA

Push back Input Stream-

A PushbackInputStream adds functionality to another input stream, namely the ability to “pushback” or “unread” one byte. One of the use of buffering is the implementation of pushback.

Pushback is used on an input stream to allow a byte to be read and then returned to the stream. Beyond the familiar methods of the InputStream class, PushbackInputStream provides **unread()** method, shown here:

```
public java.io.PushbackInputStream(java.io.InputStream, int);
```

Creates a PushbackInputStream with a pushback buffer of the specified size, and saves its argument, the input stream in, for later use. This allows multiple bytes to be returned

to the input stream.

public java.io.PushbackInputStream(java.io.InputStream);

Creates a PushbackInputStream and saves its argument, the input stream in, for later use. This allows one byte to be returned to the input stream.

public int read() throws java.io.IOException;

public int read(byte[], int, int) throws java.io.IOException;

public void unread(int) throws java.io.IOException;

Push back a byte by copying it to the front of the pushback buffer

public void unread(byte[], int, int) throws java.io.IOException;

Push back a portion of an array of bytes by copying it to the front of the pushback buffer.

public void unread(byte[]) throws java.io.IOException;

Push back an array of bytes by copying it to the front of the pushback buffer.

public int available() throws java.io.IOException;

public long skip(long) throws java.io.IOException;

public boolean markSupported();

PushbackInputStream has the side effect of invalidating the mark() or reset() methods of the InputStream used to create it. Use markSupported() method to check any stream on which you are going to use mark()/reset().

public synchronized void mark(int);

public synchronized void reset() throws java.io.IOException;

public synchronized void close() throws java.io.IOException;

Example 15.11:

```
1.      import java.io.*;
2.      class PushbackInputStreamTest
3.      {
4.          public static void main(String args[]) throws IOException
5.          {
6.              FileInputStream fin = new FileInputStream("A.txt");
7.              PushbackInputStream p1=new PushbackInputStream(fin);
8.              int ch=p1.read();
9.              System.out.println((char)ch);
10.             ch=p1.read();
11.             System.out.println((char)ch);
12.             ch=p1.read();
13.             System.out.println((char)ch);
14.             p1.unread(ch); // or p1.unread('z');
15.             ch=p1.read();
16.             System.out.println((char)ch);
17.         }
18.     }
```

Output: A.txt contains “Matrix Computers”

```
m
a
t
t
```

PushbackReader

A character stream reader that allows characters to be pushed back into the stream.

PushbackReader class allows one or more character to be returned to the input stream. This allows you to look ahead in the input stream.

public java.io.PushbackReader(java.io.Reader, int);

Creates a pushback reader with a pushback buffer of the given size. PushbackReader class provides unread() method, which returns one or more characters to the myoking input stream.

public java.io.PushbackReader(java.io.Reader);

Creates a pushback reader with a on-character pushback buffer.

public int read() throws java.io.IOException;

public int read(char[], int, int) throws java.io.IOException;

public void unread(int) throws java.io.IOException;

Push back a single character.

public void unread(char[], int, int) throws java.io.IOException;

Push back a portion of an array of characters by copying it to the front of the pushback buffer.

public void unread(char[]) throws java.io.IOException;

Push back an array by copying it to the front of the pushback buffer.

public boolean ready() throws java.io.IOException;

public void mark(int) throws java.io.IOException;

public void reset() throws java.io.IOException;

public boolean markSupported();

public void close() throws java.io.IOException;

public long skip(long) throws java.io.IOException;

Note:- An IOException will be thrown if there is an attempt to return a character when the pushback buffer is full.

Example15.12:

```
1.      import java.io.*;
2.      class PushbackReaderTest
3.      {
4.          public static void main(String args[]) throws IOException
5.          {
6.              FileReader fr = new FileReader("A.txt");
7.              PushbackReader pr=new PushbackReader(fr);
8.              int ch=pr.read();
9.              System.out.println((char)ch);
10.             ch=pr.read();
11.             System.out.println((char)ch);
12.             ch=pr.read();
13.             System.out.println((char)ch);
14.             pr.unread(ch);
15.             ch=pr.read();
16.             System.out.println((char)ch);
17.         }
18.     }
```

Output: m

a

t

t

SequenceInputStream:-

The SequenceInputStream class is a byte stream that allows you to concatenate multiple InputStreams. The construction of a sequence InputStream is different from any other InputStream. A SequenceInputStream constructor uses either a pair of InputStream classes or an Enumeration of InputStream classes as its argument:

SequenceInputStream(InputStream is1, InputStream is2)

Initializes a newly created SequenceInputStream by remembering the two arguments, which will be read in order, first **is1** and then **is2**, to provide the bytes to be read from this SequenceInputStream.

SequenceInputStream(Enumeration e)

Initializes a newly created SequenceInputStream by remembering the argument, which must be an Enumeration that produces objects whose run-time type is InputStream.

Operationally, the class fulfills read requests from the first InputStream until it runs out and then switches over to the second one. In the case of Enumeration, it will continue through all of the InputStreams until end of the last one is reached.

Example 15.13 In the following example give the file names as Command line arguments:

```
1.      import java.io.*;
2.      class SequenceInputStreamTest
3.      {
4.          public static void main(String args[]) throws IOException
5.          {
6.              FileInputStream f1 = new FileInputStream(args[0]);
7.              FileInputStream f2 = new FileInputStream(args[1]);
8.              FileOutputStream f3 = new FileOutputStream(args[2]);
9.              SequenceInputStream sis = new SequenceInputStream(f1,f2);
10.                 int c;
11.                 while((c = sis.read()) != -1)
12.                 {
13.                     f3.write(c);
14.                 }
15.                 f3.close();f1.close();f2.close();
16.             }
17.     }
```

Output:

```
java SequenceInputStreamTest A.txt B.txt C.txt
```

PrintStream:-

It is a byte stream class, which extends the FilterOutputStream. A PrintStream adds functionality to another output stream, namely the ability to print representations of various data values conveniently.

Optionally, a PrintStream can be created so as to flush automatically; this means

that the flush method is automatically invoked after a byte array is written, one of the println methods is invoked, or a newline character or byte('\n') is written.

All characters printed by a PrintStream are converted are converted into bytes using the platform's default character encoding and then written to the OutputStream or File specified in the constructor.

PrintStream supports the print() and println() methods for all types, including object. If an argument is not a simple type, the PrintStream methods will call the object's toString() method and then print the result.

Constructors:

PrintStream(File file)

Creates a new print stream, without automatic line flushing, with the specified file.

PrintStream(String fileName)

Creates a new print stream, without automatic line flushing, with the specified file name.

PrintStream(OutputStream out)

Creates a new print stream, commented to the specified OutputStream.

PrintStream(OutputStream out, boolean autoFlush)

Creates a new print stream, connected to the specified OutputStream.



Example15.14

```
1.      import java.io.*;
```

```
2.     class PrintStreamTest
3.     {
4.         public static void main (String args[]) throws IOException
5.         {
6.             PrintStream ps = new PrintStream(System.out, true);
7.             ps.println("This is a string");
8.             int i = -7;
9.             ps.println(i);
10.            double d = 4.5e-7;
11.            ps.println(d);
12.        }
13.    }
```

Output:

```
This is a string
-7
4.5E-7
```

Example 15.15

```
1.     import java.io.*;
2.     class PrintStreamTest2
3.     {
4.         public static void main (String argv[]) throws IOException
5.         {
6.             FileOutputStream fout = new FileOutputStream("test.dat");
7.             PrintStream ps = new PrintStream(fout, true);
8.             ps.println("This is a string");
9.             int i = -7;
10.            ps.println(i);
```

```
11.                double d = 4.5e-7;
12.                ps.println(d);
13.                }
14.        }
```

PrintWriter:-

Although using System.out to write to the console is still permissible under Java, its use is recommended mostly for debugging purpose or for sample programs.

For real-world programs, the recommended method of writing to the console when using java is through a PrintWriter stream. PrintWriter is one of the character-based classes. Using a character-based class for console output makes it easier to internationalize your program.

Constructors:-

PrintWriter(File file)

Creates a new PrintWriter, without automatic line flushing, for writing to the specified file.

PrintWriter(String fileName)

Creates a new PrintWriter, without automatic line flushing, for writing to the specified file.

PrintWriter(OutputStream out)

Creates a new PrintWriter, without automatic line flushing, for writing to the specified OutputStream.

PrintWriter(OutputStream out, boolean autoFlush)

Creates a new `PrintWriter` for writing to the specified `OutputStream`.

`PrintWriter(Writer out)`

Creates a new `PrintWriter`, without automatic line flushing, for writing to the specified `Writer`.

`PrintWriter(Writer out, Boolean autoFlush)`

Creates a new `PrintWriter` for writing to the specified `Writer`.

The most commonly used constructor is:

`PrintWriter(OutputStream os, boolean flushOnNewLine)`

Here, `os` is an object of type `OutputStream`, and `flushOnNewLine` controls whether Java flushes the output stream every time a `println()` method is called. If `flushOnNewLine` is true, flushing automatically takes place. If false, flushing is not automatic.

`PrintWriter` support `print()` and `println()` methods for all types including objects. If an argument is not a simple type, the `PrintWriter` methods call the object's `toString()` method and then print the result.

To write the console by using a `PrintWriter`, specify `System.out` for the output stream and flush the stream after each new line. For example, this line of code creates a `PrintWriter` that is connected to console output.

```
PrintWriter pw = new PrintWriter(System.out, true)
```

Example15.16

```
1.      import java.io.*;
2.      class PrintWriterTest
3.      {
4.          public static void main (String argv[]) throws IOException
5.          {
6.              PrintWriter pw = new PrintWriter(System.out, true);
```

```
7.         pw.println("This is a string");
8.         int i = -7;
9.         pw.println(i);
10.        double d = 4.5e-7;
11.        pw.println(d);
12.        }
13.    }
```

Output:

```
This is a string
-7
4.5E-7
```

Example 15.17

```
1.    import java.io.*;
2.    class PrintWriterTest1
3.    {
4.        public static void main (String argv[]) throws IOException
5.        {
6.            FileWriter fout = new FileWriter("test.dat");
7.            PrintWriter pw = new PrintWriter(fout, true);
8.            pw.println("This is a string");
9.            int i = -7;
10.           pw.println(i);
11.           double d = 4.5e-7;
12.           pw.println(d);
13.        }
14.    }
```

Example15.18

```
1.      import java.io.*;
2.      class PrintWriterTest2
3.      {
4.          public static void main (String argv[]) throws IOException
5.          {
6.              PrintWriter pw = new PrintWriter("test.dat");
7.              pw.println("This is a string");
8.              int i = -7;
9.              pw.println(i);
10.             double d = 4.5e-7;
11.             pw.println(d);
12.             pw.close();
13.         }
14.     }
```

PipedInputStream and PipedOutputStream:-

A piped output stream can be connected to a piped input stream to create a communications pipe. The piped output stream is the sending end of the pipe. A piped input stream should be connected to a piped output stream; the piped input stream then provides whatever data bytes are written to the piped output stream.

Typically, data is read from a PipedInputStream object by one thread and data is written to the corresponding PipedOutputStream by some other thread. Attempting to use both objects from a single thread is not recommended as it may deadlock the thread. The piped input stream contains a buffer, decoupling read operations from write operations, within limits.

Constructors:-

PipedInputStream()

Creates a PipedInputStream so that it is not yet connected.

PipedInputStream(PipedInputStream src)

Creates a PipedInputStream so that it is connected to the output stream src.

PipedOutputStream()

Creates a Piped output stream that is not yet connected to a piped input stream.

PipedOutputStream(PipedInputStream src)

Creates a piped output stream connected to the specified piped input stream.

Example 15.19

```
1.      import java.io.*;
2.      class TextGenerator extends Thread
3.      {
4.          OutputStream out;
5.          TextGenerator(OutputStream out)
6.          {
7.              this.out = out;
8.          }
9.          public void run()
10.         {
11.             try
12.             {
13.                 try
14.                 {
15.                     for(byte b = 65; b<=90; b++)
16.                         out.write(b);
17.                 }
18.                 finally
19.                 {
20.                     out.close();
21.                 }
22.             }
```

```

23.             catch(IOException e)
24.             {
25.                 System.out.println(e);
26.             }
27.         }
28.     }
29.     class Pipe
30.     {
31.         public static void main(String args[]) throws IOException
32.         {
33.             PipedOutputStream out = new PipedOutputStream();
34.             PipedInputStream in = new PipedInputStream(out);
35.             TextGenerator data = new TextGenerator(out);
36.             data.start();
37.             int ch;
38.             while((ch = in.read()) != -1)
39.                 System.out.print((char) ch);
40.         }
41.     }

```

Output:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

PipedReader and PipedWriter:-

Their functionality is same as that of PipedInputStream and PipedOutputStream with the only difference that they are character streams.

Constructors:-

PipedReader()

Creates a PipedReader so that it is not yet connected.

PipedReader(PipedWriter src)

Creates a PipedReader so that it is connected to the piped writer src.

PipedWriter()

Creates a Piped writer that is not yet connected to a piped reader.

PipedWriter(PipedReader snk)

Creates a piped writer connected to the specified piped reader.

Example 15.20

```
1.      import java.io.*;
2.      class TextGenerator extends Thread
3.      {
4.          Writer out;
5.          TextGenerator(Writer out)
6.          {
7.              this.out = out;
8.          }
9.          public void run()
10.         {
11.             try
12.             {
13.                 try
14.                 {
15.                     for(char c = 'a'; c<='z'; c++)
16.                         out.write(c);
17.                 }
18.                 finally
19.                 {
20.                     try
21.                     {
22.                         Thread.sleep(2000);
```

```

23.         }
24.         catch(Exception e) {}
25.         System.out.println("aaaaaaaaaaaaa");
26.         if(true)
27.             return;    // IOException: pipe Broken
28.         out.close(); //IOException: Write end dead
29.     }    //OK if thread finishes after closing the pipe
30.     }
31.     catch(IOException e)    { System.out.println(e);}
32. }
33. }
34. class Pipe2
35. {
36.     public static void main (String argv[]) throws IOException
37.     {
38.         PrintStream ps = new PrintStream(System.out);
39.         PipedWriter out = new PipedWriter();
40.         PipedReader in = new PipedReader(out);
41.         TextGenerator data = new TextGenerator(out);
42.         data.start(); int ch;
43.         while((ch = in.read()) != -1)
44.             System.out.print((char) ch);
45.     }
46. }

```

Output:

abcdefghijklmnopqrstuvwxyzaaaaaaaaaaaaaa

Exception in thread "main" java.io.IOException: Pipe broken

at java.io.PipedReader.read(Unknown Source)

at Pipe2.main(Pipe2.java:43)

The Filtered Byte Streams:-

The filtered streams are simply wrappers around underlying input or output streams that transparently provide some extended level of functionality. typical extensions are buffering, zip/unzip etc. The filtered byte streams are `FilterInputStream`, `FilterOutputStream`. Their constructors are:

FilterInputStream(InputStream in)

Creates a `FilterInputStream` by assigning the argument `in` to the field `this.in` so as to remember it for later use.

FilterOutputStream(OutputStream out)

Creates an output Stream filter built on top of the specified underlying output stream.

The methods provided in these classes are identical to those in `InputStream` and `OutputStream`.

Buffered Byte Streams:-

For the byte-oriented streams, a buffered stream extends a filtered stream class by attaching a memory buffer to the I/O streams. This buffer allows Java to do I/O operations on more than a byte at time, hence increasing performance.

Because, the buffer is available, skipping, marking, and resetting of the stream become possible. The buffered byte stream classes are `BufferedInputStream` and `BufferedOutputStream`. `PushbackInputStream` also implements a buffered stream.

The BufferedInputStream:-

Buffering I/O is a very common performance optimization. Java's `BufferedInputStream` class allows you to “wrap” any `InputStream` into a buffered stream and achieve this performance improvement.

`BufferedInputStream` has two constructors:

BufferedInputStream(InputStream in)

creates a `BufferedInputStream` and saves its argument, the input stream `in`, for

later use. an internal buffer array is created and stored in buf.

BufferedInputStream(InputStream in, int size)

Creates a BufferedInputStream with the specified buffer size, and saves its argument, the input stream in, for later use.

Buffering an input stream also provides the foundation required to support moving backward in the available buffer. Beyond the **read()** and **skip()** methods implemented in any Input Stream, BufferedInputStream also supports **mark()** and **reset()** methods. this support is reflected by BufferedInputStream.markSupported() returning true.



Example 15.21

```
1.      import java.io.*;
2.      class BufferedInputStreamTest
3.      {
4.          public static void main(String args[])throws IOException
5.          {
6.              FileInputStream fin=new FileInputStream("A.txt");
7.              BufferedInputStream b1 = new BufferedInputStream(fin);
8.              for(int i=1;i<=10;i++)
9.              {
10.                 System.out.print((char)b1.read());
11.             }
12.             System.out.println();
```

```

13.         if(b1.markSupported())
14.         {
15.             b1.mark(500);
16.             for(int i=1;i<=10;i++)
17.             {
18.                 System.out.print((char)b1.read());
19.             }
20.             System.out.println();
21.             b1.reset();
22.             for(int i=1;i<=10;i++)
23.             {
24.                 System.out.print((char)b1.read());
25.             }
26.         }
27.         else
28.             System.out.println("Marking option is not available");
29.         b1.close();
30.         fin.close();
31.     }
32. }

```

Output:

```

matrix Res
earch And
earch And

```

Note:

mark(32) preserves the mark for the next 32 bytes read (which is enough for all entity reference). Use of mark is restricted to access within the buffer. This means that you can only specify a parameter to mark() that is smaller than the buffer size of the stream.

BufferedOutputStream:-

Unlike buffered input, buffering output does not provide additional functionality. It is used only to improve performance.

Here are the two available constructors:

BufferedOutputStream(OutputStream out)

Creates a new buffered output stream to write data to the specified underlying output stream. it uses a buffer of size 512 bytes.

BufferedOutputStream(OutputStream out, int size)

Creates a new buffered output stream to write data to the specified underlying output stream with the specified buffer size.

Example 15.22

```
1.      import java.io.*;
2.      class BufferedOutputStreamTest
3.      {
4.          public static void main(String args[ ]) throws IOException
5.          {
6.      BufferedOutputStream f = new BufferedOutputStream (System.out,100);
7.byte buf[ ] = "This will not be displayed without flush()\n".getBytes();
8.          f.write(buf);
9.          f.write(buf);
10.         System.out.println("testing...");
11.         f.write(buf);
12.         //f.flush(); or
13.         //f.close();
14.     }
15. }
```

Output:

testing...

This will not be displayed without flush()

This will not be displayed without flush()

This will not be displayed without flush()

BufferedReader:-

The BufferedReader improves performance by buffering input. It has two constructors:

BufferedReader(Reader in)

Creates a buffering character-input stream that uses a default-sized input buffer.

BufferedReader(Reader in, int sz)

Creates a buffering character-input stream that uses an input buffer of the specified size.

Example 15.23

```
1.      import java.io.*;
2.      class BufferedReaderTest
3.      {
4.          public static void main(String args[])throws IOException
5.          {
6.              FileReader fr=new FileReader("A.txt");
7.              BufferedReader b1 = new BufferedReader(fr);
8.              for(int i=1;i<=10;i++)
9.              {
10.                 System.out.print((char)b1.read());
11.             }
12.             System.out.println();
13.             if(b1.markSupported())
14.             {
```

```

15.                b1.mark(5);
16.                for(int i=1;i<=10;i++)
17.                {
18.                    System.out.print((char)b1.read());
19.                }
20.                System.out.println();
21.                b1.reset();
22.                for(int i=1;i<=10;i++)
23.                {
24.                    System.out.print((char)b1.read());
25.                }
26.            }
27.            else
28.                System.out.println("Marking option is not available");
29.                b1.close();
30.            fr.close();
31.        }
32.    }

```

Output:

```

matrix Res
earch And
earch And

```

BufferedWriter:

Using a BufferedWriter an increase performance by reducing the number of times data is actually physically written to the output stream.

Constructors:

BufferedWriter(Writer out)

Creates a buffered character-output stream that uses a default-sized output buffer.

BufferedWriter(Writer out, int sz)

Creates a buffered character-output stream that uses an output buffer of the given size.

Example 15.24

```
1.     import java.io.*;
2.     class BufferedWriterTest
3.     {
4.         public static void main(String args[]) throws IOException
5.         {
6.     BufferedWriter f = new BufferedWriter(new PrintWriter(System.out));
7.             String s = "This will not be displayed without flush()\n";
8.             char buf[ ] = new char[s.length()];
9.             s.getChars(0,s.length(),buf,0);
10.            f.write(buf);
11.            f.write(buf);
12.            System.out.println("testing...");
13.            f.write(buf);
14.            //f.flush();
15.            //f.close();
16.        }
17.    }
```

Output:

testing...

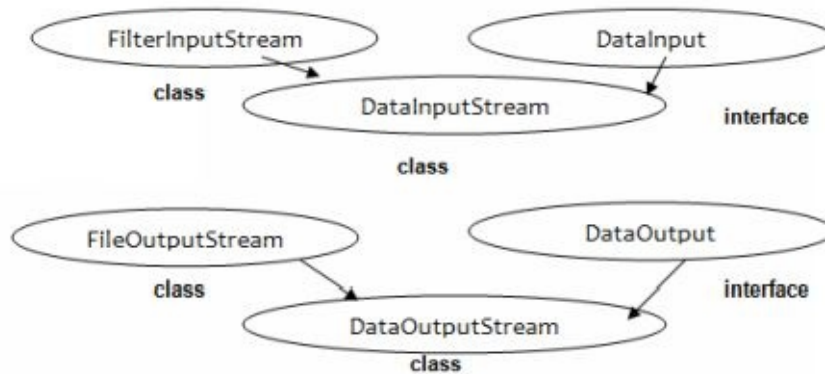
Handling Primitive Data Types Using Byte Streams:-

The basic input and output stream provide read/write methods that can only be used for reading/writing bytes or characters.

If we want to read/write the primitive data types such as integers and double, we can use filter classes as wrappers on existing input and output streams to filter data in the

original stream.

The two filter classes used for creating “data streams” for handling primitive types are `DataInputStream` and `DataOutputStream`. These classes use the concept of multiple inheritance as shown below and implement all the methods contained in both the parent class and the interface.



A data stream for input from a file can be created as follows:

```
FileInputStream fis = new FileInputStream(infile);  
DataInputStream dis = new DataInputStream(fis);
```

These statements basically wrap `dis` on `fis` and use it as a “filter”. Similarly the following statements create the output data stream `dos` and wrap it over the output file stream `fos`.

```
FileOutputStream fos = new FileOutputStream(outfile);  
DataOutputStream dos = new DataOutputStream(fos);
```

Example15.25

The following program demonstrates the reading and writing of primitive data.

1. `import java.io.*;`
2. `class ReadWritePrimitive`
3. `{`

```
4.         public static void main (String args[]) throws IOException
5.         {
6.             File primitive = new File("prim.dat");
7.             FileOutputStream fos = new FileOutputStream(primitive);
8.             DataOutputStream dos = new DataOutputStream(fos);
9.             dos.writeInt(1999);
10.            dos.writeDouble(222.44);
11.            dos.writeBoolean(false);
12.            dos.writeChar('X');
13.            dos.close();
14.            fos.close();
15.            FileInputStream fis = new FileInputStream(primitive);
16.            DataInputStream dis = new DataInputStream(fis);
17.            System.out.println(dis.readInt());
18.            System.out.println(dis.readDouble());
19.            System.out.println(dis.readBoolean());
20.            System.out.println(dis.readChar());
21.            dis.close();
22.            fis.close();
23.        }
24.    }
```

Output:

1999

222.44

false

X

Pre-defined Streams:-

System class contains three pre-defined stream variables- in, out, and err. These fields are declared as public and static within System class. This means that they can be used by any other part of your program and without reference to a specific object.

System.out refers to the standard output stream. By default, this is the console. System.out is an object of type PrintStream.

System.in refers to standard input, which is the keyboard by default. System.in is an object of type InputStream.

System.err refers to standard error stream, which also is the console by default. System.err is an object of type PrintStream.

These are byte streams, even though they typically are used to read and write characters from and to the console. You can wrap these within character-based streams, if desired.

InputStreamReader:

An InputStreamReader is a bridge from byte streams to character streams; it reads bytes and decodes them into character using a specified charset. The charset that it uses may be specified explicitly, or the platform's default charset may be accepted.

Constructors:

InputStreamReader(InputStream in)

Create an InputStreamReader that uses the default charset.

InputStreamReader(InputStream in, String charsetName)

Create an InputStreamReader that uses the named charset.

Example 15.26

```
1.      import java.io.*;
2.      class InputStreamReaderTest
3.      {
4.          public static void main (String argv[]) throws IOException
5.          {
6.              char c;
7.          InputStreamReader is = new InputStreamReader(System.in);
```

```

8.      System.out.println("Enter characters, 'q' to quit");
9.          c =(char)is.read();
10.         while(c!='q')
11.         {
12.             System.out.print(c);
13.             c =(char)is.read();
14.         }
15.     }
16. }

```

Output:

```

Enter characters, 'q' to quit
abcd
abcd
q

```

Example15.27

Previous example is modified to make use of buffering to improve performance.

```

1.      import java.io.*;
2.      class BufferedReaderTest
3.      {
4.          public static void main (String args[]) throws IOException
5.          {
6.              char c;
7.              BufferedReader br = new BufferedReader(new InputStreamReader
8.                  (System.in));
9.              System.out.println("Enter characters, 'q' to quit");
10.             c =(char)br.read();
11.             while(c!='q')
12.             {

```

```

12.                System.out.print(c);
13.                c = (char)br.read();
14.                }
15.            }
16.        }

```

Output:

Enter characters, 'q' to quit

abcd

abcd

q

Example 15.28

The InputStreamReader does not provide readLine() method, hence InputStreamReader is linked to BufferedReader that provides the readLine() method.

```

1.    import java.io.*;
2.    class BufferedReaderTest1
3.    {
4.        public static void main (String argv[]) throws IOException
5.        {
6.    BufferedReader br = new BufferedReader(new InputStreamReader (System.in));
7.        System.out.println("Enter 'stop' to quit");
8.        String str = br.readLine();
9.        while(!str.equals("stop"))
10.       {
11.           System.out.println("Your Name:"+str);
12.           str = br.readLine();
13.       }
14.   }
15. }

```

Output:

```
Enter 'stop' to quit
Harry Feat
Your Name:Harry Feat
stop
```

OutputStreamWriter:

An `OutputStreamWriter` is a bridge from character streams to byte streams: characters written to it are encoded into bytes using a specified charset. The charset that it uses may be specified explicitly, or the platform's default charset may be accepted.

Constructors:

`OutputStreamWriter(OutputStream out)`

Create an `OutputStream` that uses the default character encoding.

`OutputStreamWriter (OutputStreamout, String charsetName)`

Create an `OutputStream` that uses the named charset.

Reading Console Input:

In Java 1.0 the only way to perform console input was to use a byte stream, and older code that uses this approach persists.

Today, using a byte stream to read console input is still technically possible, but doing so may require the use of a deprecated method, and this approach is not recommended.

In Java, console input is accomplished by reading from `System.in`. To obtain a character-based stream that is attached to console, you wrap `System.in` in an `InputStreamReader` object, which can be further wrapped in a `BufferedReader` object, to improve performance.

Example 15.29

This example demonstrates that we can read directly from System.in.

```
1.     import java.io.*;
2.     class ReadFromConsole
3.     {
4.         public static void main (String arg[]) throws IOException
5.         {
6.             char c;
7.             System.out.println("Enter characters, 'q' to quit");
8.             c = (char) System.in.read();
9.             while(c!='q')
10.            {
11.                System.out.print(c);
12.                c = (char) System.in.read();
13.            }
14.        }
15.    }
```

Output:

Enter characters, 'q' to quit

abcd

abcd

xyzq

xyz

The proper way of reading from console is to wrap the System.in in InputStreamReader and to further wrap it to BufferedReader to improve performance and to input one line at a time. The appropriate examples have already been covered in the earlier section related to InputStreamReader class.

Writing Console Output:

Console output is most easily accomplished with print() and println(). These

methods are defined by the class `PrintStream` (which is the type of the object referenced by `System.out`). Even though `System.out` is a byte stream, using it for simple program output is still acceptable.

However, a character-based alternative is the right choice. Because `PrintStream` is an output stream derived from `OutputStream`, it also implements the low-level method `write()`. Thus, `write()` can be used to write to the console. The simplest form of `write()` defined by `PrintStream` is:

`void write(int byteval)`

This method writes to the stream, the byte specified by `byteval`. Although `byteval` is declared as an integer, only the low-order eight bits are written.

Example 15.30

```
1.      import java.io.*;
2.      class WriteTest
3.      {
4.          public static void main (String argv[ ]) throws IOException
5.          {
6.              int b;
7.              b = 'A';
8.              System.out.write(b);
9.              // System.out.flush();to flush
10.             System.out.write('\n');//to flush
11.         }
12.     }
```

Output:

A

Note:-

You will not often use `write()` to perform console output, because `print()` and

println() are substantially easier to use.

Serialization:

Serialization is the process of writing the state of an object to a byte stream. This is useful when you want to store the state of an object to a byte stream. This is also useful when you want to save the state of your program to a persistent storage area, such as file. At a later time, you may restore these objects by using the process of deserialization.

Serialization is also needed to implement RMI. RMI allows a java object on one machine to invoke a method of a java object on a different machine. An object may be supplied as an argument to the remote method. The sending machine serializes the object and translates it. The receiving machine deserializes it.

If you attempt to serialize an object at the top of an object graph, all of the other referenced objects are recursively located and serialized.

Similarly, during the process of deserialization, all of these objects and their references are correctly stored. Variables that are declared as transient or static are not saved by the serialization facilities.

Externalizable interface:

The Java facilities for serialization and deserialization have been designed so that much of the work to save and restore the state of an object occurs automatically.

However, there are cases in which the programmer may need to have control over these processes. For example, it may be desirable to use compression or encryption techniques. The externalizable interface is designed for these situations. The objects of class implementing Externalizable interface can also be serialized and deserialized.

But the control is with the user i.e. user can decide which part to serialize and deserialize. The serialization code is written in readExternal() method and deserialization code is written in writeExternal() method.

Methods in Externalizable interface:

void readExternal(ObjectInput in)

The object implements the readExternal method to restore its contents by calling the methods of DataInput for primitive types and readObject for object, strings and arrays.

void writeExternal(ObjectOutput out)

The object implements the writeExternal method to save its contents by calling the methods of DataOutput for its primitive values or calling writeObject method of ObjectOutput for objects, strings and arrays.

ObjectOutput Interface:

The ObjectOutput interface extends the DataOutput interface and supports object serialization. Beside methods defined in DataOutput interface, objectOutput interface define only method.

Void writeObject(Object object)

This is called to serialize an object. All methods of ObjectOutput interface will throw an IOException on error conditions.

ObjectOutputStream:

The ObjectOutputStream class extends the OutputStream class and implements the ObjectOutput interface. It is responsible for writing objects to a stream.

Constructor:

ObjectOutputStream(OutputStream out)

Creates an ObjectOutputStream that writes to the specified OutputStream. The argument out is the output stream to which serialized objects will be written.

Method to write object:

void writeObject(Object obj)

Write an object to the underlying storage or stream.

ObjectInput Interface:-

The ObjectInput interface extends the DataInput interface. It supports object serialization. Beside methods defined in DataInput interface, ObjectInput interface defines only one method:

Object readObject()

This is called to deserialize an object.

ObjectInputStream:

The ObjectInputStream class extends the InputStream class and implement the ObjectInput interface. ObjectInputStream is responsible for reading object from a stream.

Constructor:

ObjectInputStream(InputStream in)

Creates an ObjectInputStream that reads from the specified InputStream. The argument in, is the input stream from which serialized objects should be read.

The method for deserializing is :

Object readObject()

Read an object from the ObjectInputStream.

Example 15.31

```
1.      import java.io.*;
2.      class SerializationTest
3.      {
4.          public static void main (String args[]) throws IOException
5.          {
```

```
6.         try
7.         {
8.             MyClass object1 = new MyClass("Hello", 123, 1.3e3);
9.             MyClass object2 = new MyClass("Hello1", 1, 2);
10.            System.out.println(object1);
11.            System.out.println(object2);
12.    FileOutputStream fos = new FileOutputStream("serial.dat");
13.    ObjectOutputStream oos = new ObjectOutputStream(fos);
14.            oos.writeObject(object1);
15.            oos.writeObject(object2);
16.            oos.flush();
17.            oos.close();
18.        }
19.        catch(Exception e)
20.        {
21.            System.out.println(e);
22.        }
23.        try
24.        {
25.            MyClass object3, object4;
26.    FileInputStream fis = new FileInputStream("serial.dat");
27.    ObjectInputStream ois = new ObjectInputStream(fis);
28.            object3 = (MyClass) ois.readObject();
29.            object4 = (MyClass) ois.readObject();
30.            ois.close();
31.            System.out.println("After deserialization");
32.            System.out.println(object3);
33.            System.out.println(object4);
34.        }
35.        catch(Exception e)
36.        {
37.            System.out.println(e);
```

```

38.         }
39.     }
40. }
41. class MyClass implements Serializable
42. {
43.     String s;
44.     int i=100;
45.     double d;
46.     MyClass(String s, int i, double d)
47.     {
48.         this.s = s;
49.         this.i = i;
50.         this.d = d;
51.     }
52.     public String toString()
53.     {
54.         return "s = "+ s +"; i = "+ i + " ; d = " +d;
55.     }
56. }

```

Output:

s = Hello; i = 123 ; d = 1300.0

s = Hello1; i = 1 ; d = 2.0

After deserialization

s = Hello; i = 123 ; d = 1300.0

s = Hello1; i = 1 ; d = 2.0

Note: we can save more than one object of the same type in a file and can also save object of different types in the same file.

Example15.32

This example demonstrates that the transient and static variables are not persisted during serialization.

```
1. import java.io.*;
2. class SerializationTest2
3. {
4.     public static void main (String args[]) throws IOException
5.     {
6.         try
7.         {
8.             MyClass object1 = new MyClass("Hello", 123, 1.3e3);
9.             MyClass object3 = new MyClass("Hello1", 1, 2);
10.            object3.x = 200;
11.            A object5 = new A();
12.            System.out.println(object1);
13.            System.out.println(object3);
14.            System.out.println(object5);
15.            FileOutputStream fos = new FileOutputStream("serial.dat");
16.            ObjectOutputStream oos = new ObjectOutputStream(fos);
17.            oos.writeObject(object1);
18.            oos.writeObject(object3);
19.            oos.writeObject(object5);
20.            object3.x = 400;
21.            oos.flush();
22.            oos.close();
23.        }
24.        catch(Exception e)
25.        {
26.            System.out.println(e);
27.        }
28.        try
29.        {
30.            MyClass object2, object4;
31.            A object6;
32.            FileInputStream fis = new FileInputStream("serial.dat");
```

```
33.    ObjectInputStream ois = new ObjectInputStream(fis);
34.        object2 = (MyClass) ois.readObject();
35.        object4 = (MyClass) ois.readObject();
36.        object6 = (A) ois.readObject();
37.        ois.close();
38.        fis.close();
39.        System.out.println("After deserialization");
40.        System.out.println(object2);
41.        System.out.println(object4);
42.        System.out.println(object6);
43.    }
44.    catch(Exception e)
45.    {
46.        System.out.println(e);
47.    }
48.    }
49. }
50. class MyClass implements Serializable
51. {
52.     String s;
53.     transient private int i=100;
54.     public static int x = 10;
55.     double d;
56.     MyClass(String s, int i, double d)
57.     {
58.         this.s = s;
59.         this.i = i;
60.         this.d = d;
61.         x = 100;
62.     }
63.     public String toString()
64.     {
```



```

65.             return "s = " + s + "; i = "+ i + " ;d = " + d +"; x = "+ x;
66.         }
67.     }
68.     class A implements Serializable
69.     {
70.         public String toString()
71.         {
72.             return "Class A";
73.         }
74.     }

```

Example 15.33

Using Externalizable

```

1.     import java.io.*;
2.     class SerializationTest1
3.     {
4.         public static void main (String args[]) throws IOException
5.         {
6.             try
7.             {
8.                 MyClass object1 = new MyClass("Hello", 99, 123.45);
9.                 System.out.println(object1);
10.            FileOutputStream fos = new FileOutputStream("serial.dat");
11.            ObjectOutputStream oos = new ObjectOutputStream(fos);
12.                System.out.println("before writing object");
13.                oos.writeObject(object1);
14.                System.out.println("after writing object");
15.                oos.flush();
16.                oos.close();
17.            }

```

```
18.         catch(Exception e)
19.         {
20.             System.out.println(e);
21.         }
22.     try
23.     {
24.         MyClass object2;
25.         FileInputStream fis = new FileInputStream("serial.dat");
26.         ObjectInputStream ois = new ObjectInputStream(fis);
27.         System.out.println("before reading object");
28.         object2 = (MyClass) ois.readObject();
29.         System.out.println("after reading object");
30.         ois.close();
31.         System.out.println(object2);
32.     }
33.     catch(Exception e)
34.     {
35.         System.out.println(e);
36.     }
37. }
38. }
39. class MyClass implements Serializable
40. {
41.     String s;
42.     int i=100;
43.     double d;
44.     public MyClass() {};// It is must
45.     public MyClass(String s, int i, double d)
46.     {
47.         this.s = s;
48.         this.i = i;
49.         this.d = d;
```

```

50.         }
51.         public String toString()
52.         {
53.             return "s = "+ s +"; i = "+ i + " ;d = " +d;
54.         }
55.     public void writeExternal(ObjectOutput oos) throws IOException
56.     {
57.         System.out.println("Inside writeExternal");
58.         oos.writeObject(s);
59.         oos.writeInt(i);
60.         oos.writeDouble(d);
61.     }
62. public void readExternal(ObjectInput ois) throws ClassNotFoundException,
        IOException
63.     {
64.         System.out.println("Inside readExternal");
65.         s = (String) ois.readObject();
66.         i = ois.readInt();
67.     }
68.     }

```

Output:

```

s = Hello; i = 99 ;d = 123.45
    before writing object
    after writing object
    before reading object
    after reading object
s = Hello; i = 99 ;d = 123.45

```

Note:

During deserialization, object will be constructed using default constructor (unlike serializable interface) and then the initialization will take place. Testoring state should be done in readExternal(). Similarly during serialization only class identification is saved. Any state must be saved explicitly in writeExternal().





CHAPTER

∞ 16 ∞

(Applet)

Introduction-

An applet is an application designed to travel over the Internet and to be executed on the client machine by a Java Compatible web browser like Internet Explorer or Netscape Navigator. Applets are also Java programs but they reside(stored) on the servers.

An applet cannot be executed like standalone application. Applet can be executed only by embedding it into an HTML page like an image or sound file. To run an applet we need to access an HTML page which has applet embedded into it. When the web browser downloads such an HTML page, it subsequently loads the executable file, which contains applet code and then executes it on the local machine.

After an applet arrives on the client, it has limited access to resources, so that it can produce an arbitrary multi-media user interface and run complex computations without introducing the risk of viruses or breaching (breaking) data integrity.

Applet Architecture:

Applets are different from normal Java programs.

- ✚ Applets are GUI based (window-based) programs
- ✚ Applets are event-driven

When a normal Java program (which is not GUI based) needs input it prompts the

user and then calls some input method, such as `readLine()` i.e. the interaction is initiated by the program.

This is not the way in which GUI-based programs behave. The user initiates interaction with the program rather than program initiating the action.

For example, in a word processing software, user initiates action by clicking on different buttons, which generates an event and some piece of code is executed as a result and accordingly some action takes place.

Applets use `awt` package (Abstract Windows Toolkit) for providing GUI and for event-handling. The `awt` is called so because it totally depends on the functionality of the underlying operating system.

An applet resembles a set of interrupt service routines. An applet waits until an event occurs. The `awt` notifies the applet about an event by calling an event handler that has been provided by the applet.

Once this happens, the applet must take appropriate action and then quickly return control to the AWT.

This is a crucial point. For the most part, our applet should not enter a “mode” of operation in which it maintains control for an extended period. Instead, it must perform specific actions in response to events and then return control to the AWT run time system.

In those situations in which our applet needs to perform a repetitive task on its own (for example, displaying a scrolling message across its window), we must start an additional thread of execution.

Writing an Applet:

All applets are sub classes of `Applet` class. The `Applet` class is contained in the package `java.applet`. The hierarchy of the `Applet` class is as follows:

```
java.lang.Object
    |
    java.awt.Component
        |
        java.awt.Container
            |
            java.awt.Panel
                |
                java.applet.Applet
```

The `Applet` class provides all necessary support for applet execution, such as starting and stopping.

It also provides methods that load and display images. It also provides necessary support for all the window-based activities.

Our sub class extending the applet class must always be declared as public as it is instantiated (creating object) and executed by the web browser.

The browser makes use of no argument constructor when instantiating the applet so it is must to provide no-argument public constructor with the public visibility.

It is recommended that we do not provide constructor in the Applet class as in that case compiler will provide the default no argument constructor.

We can make use of the `init()` method for initialization. While writing applet code we normally override some of the methods of the Applet class, which are invoked automatically during applet execution.

It is very common to override the `paint()` method. The code in the `paint()` method mainly displays the output in the Applet window.

Example 16.1 (App1.java) The following program illustrates a simple applet, which just displays “Hello World” inside a window. Whatever we draw/display in the applet’s `paint` method, it appears in the applet’s window.

In the following example, the background color of the applet window is set to red and the foreground color is set to green so the background would appear as red and text “Hello Word” would be displayed in green color.

```
1.      import java.awt.*;
2.      import java.applet.Applet;
3.      /*<applet code= “App1” width =500 height=100></applet>*/
4.      public class App1 extends Applet
5.      {
6.          public void init()
7.          {
8.              setBackground(Color.red);
9.              setForeground(Color.green);
10.         }
11.         public void paint(Graphics g)
12.         {
13.             g.drawString(“Hello World”,20,20);
14.         }
15.     }
```

Output:



Executing an Applet:

Applets are not executed by the console-based Java run-time interpreter. There are two ways in which we can run an applet .

1. Executing the applet within a Java compatible web browser.
2. Using an appletviewer, such as the standard SDK tool, appletviewer. An appletviewer executes our applet in a window. This is fastest and easiest way to test our applet.

Executing Applet in a web browser:

We need to write a short HTML text file that contains the appropriate APPLET tag. The applet tag must include at least following three attributes:

Code

Width

Height

The attribute code's value specifies the name of the class containing applet's code. The attribute width and height specify the width and height of the applet's windows respectively.

Here the HTML file **RunApp.Html** with applet "**App1**" embedded into it.

```
<html>
  <head>
    <title>Simple Applet</title>
  </head>
  <body>
    <h1>Simple "Hello World" Application </h1>
```



```
<applet code= "App1" width =500 height=100>
</applet>

</body>
</html>
```

You can execute the applet by opening file **RunApp.html** in web browser. The file can be on local file system or can also be loaded from a web server.

Executing Applet using appletviewer:

However, a more convenient method exists that we can use to speed up testing. Simply include a comment in your java source code file that contains that APPLET tag. By doing so, our code is documented with a prototype of the necessary HTML statements, and we can test your compiled output merely by starting the appletviewer with our Java source code file. if we use this method, then after compiling we can execute the applet as follows:

```
appletviewer SimpleApplet.Java
```

Applet Life cycle Methods:

Applet overrides a set of methods that provides the basic mechanism by which the browser or appletviewer interfaces to the applet and controls its execution.

These methods are also called lifecycle methods because the browser or applet viewer calls them automatically during different stages of applet lifecycle. In all there are five lifecycle methods:

```
public void init ()
public void start ()
public void stop ()
public void destroy ()
public void paint (Graphics g)
```

Four of these methods: init(), start(), stop(), and destroy() are defined in the applet class.

The paint(), is defined by the awt Component class, default implementations for all these methods are provided. Applets do not need to override those methods they do not use. However, very simple applets will not need to define them.

Applet Initialization and Termination:

When an applet begins, the following methods are called in sequence

- 1. init()**

2. **start()**

3. **paint()**

When an applet is terminated, the following sequence of method calls takes place:

4. **stop()**

5. **destroy()**

The init() Method:

This is the first method to be called. This is where we should initialize variables and write code for other initialization activities. This method is called only once during the life cycle of our applet immediately after instantiation.

The start() Method:

The start() method is called after init(). It is also called to restart an applet after it has been stopped. Where init() is called once the first time an applet is loaded, start() is called each time an applet's HTML document is displayed on screen. So, if a user leaves a web page and comes back, the applet resumes execution at start().

The paint() Method:

The paint() is called after the init() and start() method when the applet begins execution. The paint() method is also called each time our applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. or the applet window may be minimized and then restored.

The Stop() Method:

The stop() method is called when a web browser leaves the HTML document containing the applet when it goes to another page, for example. When stop() is called, the applet is probable running.

We should use stop() to suspend threads that do not need to run when the applet is not visible. We can restart them when start() is called if the user returns to the page. We can also free any costly resource and acquire it again in the start() method.

The destroy () Method:

The destroy () method is called when the environment determines that our applet needs to be removed completely from memory. At this point, we should free up any

resources the applet may be using. The stop() method is always called before destroy().

Example 16.2 A simple applet that sets the background color to cyan, the foreground color to red, and displays a message that illustrates the order in which the init(), start(), and paint() methods are called when an applet starts up.

```
1.      import java.awt.*;
2.      import java.applet.*;
3.      /*<applet code= "App2" width =500 height=100></applet>*/
4.      public class App2 extends Applet
5.      {
6.          String msg="";
7.      public void init()
8.          {
9.              System.out.println("Inside init");
10.             setBackground(Color.cyan);
11.             setForeground(Color.red);
12.             msg = "Inside init ()—";
13.         }
14.     public void start()
15.         {
16.             System.out.println ("Inside start");
17.             msg += "Inside start ()—";
18.         }
19.     public void stop()
20.         {
21.             System.out.println("Inside Stop");
22.             msg += "Inside stop ()—";
23.         }
24.     public void paint (Graphics g)
25.         {
26.             System.out.println ("Inside paint");
27.             msg += "Inside paint ()—";
28.             g.drawString(msg,10,30);
29.         }
```

```

30.         public void destroy()
31.         {
32.             System.out.println ("Inside destroy");
33.         }
34.     }

```

Output:



Dynamic Applet:

The output displayed in the Applet's window can be made dynamic. For example, we can display a moving banner. The applet can also be used in applications like displaying score and other statistics related to a Cricket match by retrieving from server; displaying stock prices etc.

Example16.3

The following example displays a moving banner. It demonstrates use of repaint() method to refresh the display. This applet scrolls a message, from right to left, across the applet's window.

Since the scrolling of the message is a repetitive task, it is performed by a separate thread, created by the applet when it is initiated. This program also demonstrates that the resources can be freed in the stop() method and can be re acquired in the start() method.

```

1.     import java.awt.* ;
2.     import java.applet.*;
3.     /*<applet code="App3" width=500 height=100></applet>*/
4.     public class App3 extends Applet implements Runnable
5.     {
6.         String msg="A Simple Moving Banner." ;
7.         Thread t = null;
8.         boolean stopFlag;
9.         public void init ()
10.        {

```

```
11.         setBackground(Color.cyan);
12.         setForeground(Color.red);
13.     }
14.     public void start()
15.     {
16.         t=new Thread(this);
17.         stopFlag = false ;
18.         t.start ();
19.     }
20.     public void stop()
21.     {
22.         stopFlag=true;
23.         t=null;
24.     }
25.     public void paint(Graphics g)
26.     {
27.         g.drawString(msg,50,30);
28.     }
29.     public void run()
30.     {
31.         for (;;)
32.         {
33.             repaint();
34.             try
35.             {
36.                 Thread.sleep(250);
37.             }
38.             catch(InterruptedException e)
39.             {
40.                 System.out.println(e);
41.             }
42.             msg = msg.substring (1) + msg.charAt(0);
```

```

43.                 if (stopFlag)
44.                     break;
45.                 }
46.             }
47.         }

```

Output:



Using the status window:

In addition to displaying information in its window, an applet can also output a message to the status windows of the browser or appletviewer on which it is running.

To do so call `showStatus()` with the string that we want to display. The status window is a good place to give the user feedback about what is occurring in the applet, suggest options, or possibly report some types of errors.

The status window also makes an excellent debugging aid (help), because it gives us an easy way to output information about our applet.

Example 16.4: The following example demonstrate how to use of the status window.

```

1.     import java.awt.*;
2.     import java.applet.*;
3.     /*<applet code="App4" width=500 height=100></applet>*/
4.     public class App4 extends Applet
5.     {
6.         public void init( )
7.         {
8.             setBackground(Color.cyan);
9.         }
10.        public void paint(Graphics g)
11.        {
12.            g.drawString("This is in the applet window",10,20);

```

```
13.     showStatus("This is shown in the status window");
14.         }
15.     }
```

Output:



The HTML's APPLETTAG:

The APPLETTAG is used to start an applet from both an HTML document and from an appletviewer.

An appletviewer will execute each APPLETTAG that it finds in a separate window; while web browsers like Netscape Navigator, and Internet Explorer allow many applets on a single page.

The syntax for the standard APPLETTAG is :

< APPLETTAG

 CODE=applet class name

 CODEBASE=codebase URL

 ALT=alternate Text,

 NAME=appletInstancename

 WIDTH=pixels

 HEIGHT=pixels

 ALIGN=alignment

 VSPACE=vertical space in pixels

 HSPACE=horizontal space in pixels

>

<PARAM NAME=attributeName-1 VALUE=Attribute Value-1>

<PARAM NAME=attributeName-2 VALUE=Attribute Value-2>

<PARAM NAME=attributeName-N VALUE=Attribute Value-N>

.....HTML displayed in the absence of Java.....

</APPLET>

CODE:

CODE is a required attribute that gives the name of the file containing our applet's compiled class file. This file is relative to the CODEBASE URL of the applet, which is the directory that the html file was in or directory indicated by CODEBASE if set.

CODEBASE:

CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag).

The HTML document's URL directory is used as the CODEBASE if this attribute is not specified. The CODEBASE does not have to be on the host from which the HTML document was read.

ALT: The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can not currently run Java applet. This is distinct from the alternate HTML we provide for browsers that do not support applets.

NAME: NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them. To obtain an applet by name, use **getApplet()**, which is defined by the Applet Context interface.

WIDTH and HEIGHT: Size of applet's display area in pixels.

ALIGN: ALIGN is an optional attributes that specifies the alignment of the applet. This attribute is treated the same way as the HTML'S IMG tag with following possible values:

LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, AND ABSBOTTOM.

VSPACE and HSPACE: These attributes are optional VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels on each side of the applet. They are treated the same as IMG tag's VSPACE and HSPACE attributes.

The PARAM Tag

The PARAM tag allows we to specify applet specific arguments in an HTML page. Applets access their attributes with the **getParameter()** method. PARAM tag has two attributes : NAME and VALUE.

Example 16.5 This example demonstrates how to pass parameters to Applet from the HTML page.

```
1.      import java.awt.*;
2.      import java.applet.*;
3.      /*<applet code="App5" width=500 height=100>
5.      <param name="fontName" value="Arial">
6.      <param name="fontSize" value="30">
7.      </applet>*/
8.      public class App5 extends Applet
9.      {
10.         String fName;
11.         int fSize;
12.         public void init ( )
13.         {
14.             fName = getParameter("fontName");
15.             if (fName == null)
16.                 fName = "Courier";
17.             fSize = Integer.parseInt(getParameter("fontSize"));
18.             Font f1= new Font(fName, Font.PLAIN, fSize);
19.                 setFont(f1);
20.         }
21.         public void paint (Graphics g)
22.         {
23.             g.drawString ("Font Name : " + fName, 0, 25 );
24.                 g.drawString ("Font Size : " + fSize, 0, 50 );
25.                 g.drawString ("Hello world", 0,75);
26.         }
27.     }
```

Output:



Getting DocumentBase and CodeBase:

Often, we will create applets that will need to explicitly load media and text. Java will allow the applet to load data from the directory holding the HTML file that started the applet (the document base) and the directory from which the applet's class file was loaded (the code base).

These directories are returned as URL objects. They can be concatenated with a string that name the file we want to load. To actually load another file, we will use the `showdocument()` method defined by the Applet context interface.

Example 16.6 The following example demonstrates how we can obtain the document base and code base in the applet code.

App6.java

```
1.      import java.awt.*;
2.      import java.applet.*;
3.      import java.net.URL;
4.      public class App6 extends Applet
5.      {
6.          public void paint (Graphics g)
7.          {
8.              URL u1 = getCodeBase( );
9.              g.drawString ("Code base:-" + u1, 10,20);
10.             URL u2 = getDocumentBase();
11.             g.drawString ("Document base:-"+u2, 10,40);
```

12. }

13. }

App6.html

1. <html>

2. <body>

3. <applet code="App6" width=500 height=200></applet>

4. </body>

5. </html>

Output:

AppletContext and Show Document():

One application of Java is to use active image and animation to provide a graphical means of navigating the web that is more interesting than the underlined blue words used by hypertext. To allow our applet to transfer control to another URL, we must use the `showDocument()` method defined by the Applet Context interface.

The Applet Context is an interface that lets us get information from the applet's execution environment. The context of the currently executing applet is obtained by a call to the `getAppletContext()` method defined by the applet class.

Within an applet, once we have obtained the applet's context, we can bring another document into view by calling `showDocument()` method. This method has no return value and returns no exception if it fails so use it carefully. There are two `showDocument()` methods.

void ShowDocument(URL url)

Replaces the Web page currently being viewed with the given URL

void ShowDocument (URL url, String where)

Displays the specified document at the specified location within the browser window. Valid arguments for “where’ are:

- “_self”(show in the current frame)**
- “_parent” (show in the parent frame)**
- “_top” (show in topmost frame), and**
- “_blank”(show in new browser window)**

Example 16.7

The following applet demonstrates use of Applet Context and showDocument(). Upon execution, it obtains the current applet context and uses that context to transfer control to a file called A.html. This file must be in the same directory as the applet. Test.html can contain any valid hypertext that your like.

```
1.      import java.awt.*;
2.      import java.applet.*;
3.      import java.net.*;
4.      public class App7 extends Applet
5.      {
6.          AppletContext ac;
7.          URL u1;
8.          public void init()
9.          {
10.             setBackground(Color.cyan);
11.          }
12.      public void start( )
13.      {
```

```
14.         ac = getAppletContext();
15.         u1 = getCodeBase();
16.         try
17.         {
18. Thread.sleep(3000);
19. ac.showDocument(new URL(u1 + "A.html"), "_blank");
20.         }
21.         catch(MalformedURLException e)
22.         {
23.             showStatus("Invalid URL");
24.             e.printStackTrace() ;
25.         }
26.         catch(InterruptedException e)
27.         {
28.             showStatus("Invalid URL");
29.             e.printStackTrace () ;
30.         }
31.     }
32.     public void paint (Graphics g)
33.     {
34.         g.drawString(u1+"A.html",20,20);
35.         try
36.         {
37.             Thread.sleep(3000);
38.         }
39.         catch(InterruptedException e)
40.         {
41.             System.out.println(e);
42.         }
43.     }
44. }
```

App7.html

```
1.      <html>
2.          <head>
3.              <Title>Applet Context Demo</Title>
4.          </head>
5.          <body>
6.              <applet code="App7" width=200 height=200></applet>
7.          </body>
8.      </html>
```

A.html

```
1.      <html>
2.          <head>
3.              <title>New Page</title>
4.          </head>
5.          <body bgcolor=blue text=red>
6.              <h1>matrix</h1>
7.          </body>
8.      </html>
```

Example16.8

```
1.      import java.applet.*;
2.      import java.awt.*;
3.      import java.awt.event.*;
4.      /*<Applet code="CopyApp" width=500 height=200></applet>*/
5.      public class CopyApp extends Applet implements ActionListener
6.      {
7.          TextField tf1,tf2;
8.          Button b1;
```

```
9.         public void init()
10.        {
11.            setBackground(Color.red);
12.            Font f1=new Font("Arial",
13.            Font.BOLD+Font.ITALIC,20);
14.            tf1=new TextField(8);
15.            tf2=new TextField(8);
16.            tf1.setFont(f1);
17.            tf2.setFont(f1);
18.            b1=new Button("Copy");
19.            b1.addActionListener(this);
20.            b1.setFont(f1);
21.            add(tf1);
22.            add(tf2);
23.            add(b1);
24.        }
25.        public void paint(Graphics g)
26.        {
27.            g.drawString("Matrix",0,150);
28.        }
29.        public void actionPerformed(ActionEvent ae)
30.        {
31.            String s1=tf1.getText();
32.            tf2.setText(s1);
33.        }
34.    }
```

Output:



Example 16.9 Sum of 2 numbers using Applet

```
1.      import java.applet.*;
2.      import java.awt.*;
3.      import java.awt.event.*;
4.      /*<applet code = "SumApplet" width=400 height=400></applet> */
5.      public class SumApplet extends Applet implements ActionListener
6.      {
7.          TextField tf1,tf2,tf3;
8.          Label l1,l2,l3;
9.          Button b1;
10.         public void init()
11.         {
12.             setLayout(new GridLayout(4,2));
13.             l1 = new Label("No. 1");
14.             l2 = new Label("No. 2");
15.             l3 = new Label("Result");
16.             tf1 = new TextField(8);
17.             tf2 = new TextField(8);
18.             tf3 = new TextField(8);
19.             b1 = new Button("Sum");
20.             b1.addActionListener(this);
21.             add(l1);add(tf1);
22.             add(l2);add(tf2);
23.             add(l3);add(tf3);
```



```
24.         add(b1);
25.     }
26.     public void actionPerformed(ActionEvent ae)
27.     {
28.         Button abc = (Button)ae.getSource();
29.         if(abc == b1)
30.         {
31.             int a,b,c;
32.             a = Integer.parseInt(tf1.getText());
33.             b = Integer.parseInt(tf2.getText());
34.             c = a + b;
35.             tf3.setText(String.valueOf(c));
36.         }
37.     }
38. }
```





CHAPTER

∞ 17 ∞

(Abstract Windows Toolkit - AWT)

Introduction-

Java provides a large number of built-in classes, which help us in designing graphical user interface (GUI). Most of these classes belong to the package `java.awt` and are collectively known as Abstract Windows Toolkit (AWT).

We are familiar with software like MS-WORD, MS-EXCEL etc. All of these software have a GUI i.e. when we run these software, we see graphical user interface.

We access various features of this software by selecting different options through keyboard or mouse. Lot of programming is needed to display the user interface.

Three important parts of AWT, which help in designing graphical user interfaces, are

- **Components**
- **Containers**
- **Layout Managers**

All of the above play an important role in designing the GUI.

Components:

Components are Java's building blocks for creating GUI's. A component is an object having graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, check boxes and scrollbars of a typical graphical user interface. The `java.awt.Component` class is the abstract super class of the non-menu related AWT components. Class component can also be extended directly to create lightweight component. A lightweight is a component that is not associated with a native opaque window.

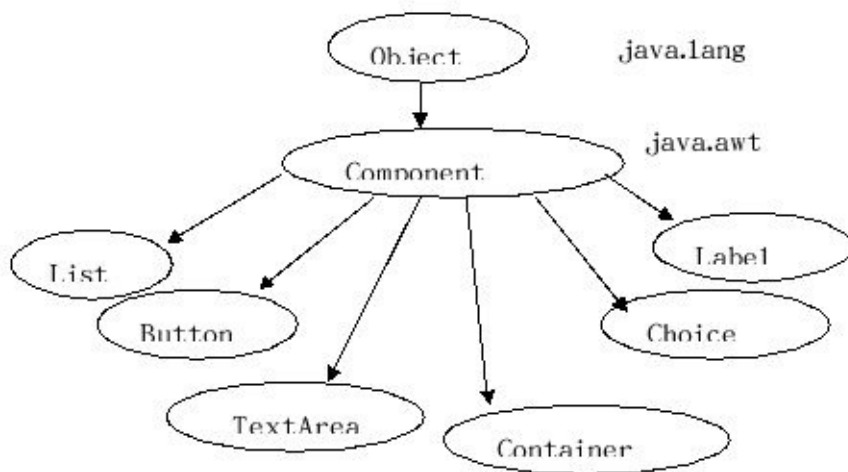
Some components types, such as buttons and scrollbars, are used directly for GUI control. Other kinds of components (those that inherit from the `Container` class) provide special organization. While the components like `TextField`, `TextArea`, `List` etc, are used to accept input from the user. The label is a different type of component whose purpose is to simply display the appropriate label to identify other components.

Java's components are implemented by the many subclasses of the `java.awt.Component` and `java.awt.Menu` component super classes.

One way to organize this fairly large number of classes is to divide them into categories:

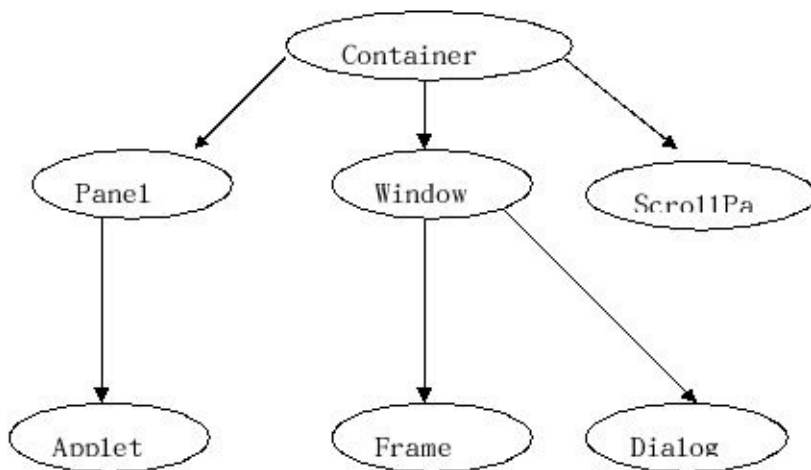
- **Visual components**
- **Container components**
- **Menu components**

Component hierarchy: The following diagram shows AWT the component hierarchy.



Containers:

The following diagram describes the container hierarchy



All the window-based applications start with a top-level window visible on the screen.

Similarly all GUI based java applications also start with a top-level window, which is referred to as top-level container in java's terminology.

Containers are also java components that can contain other components. A component can be made visible only by adding it to a container or putting it inside a container.

All java GUI's reside either in an applet or in a Frame. The applets and frames are the two top-level containers in java.

The applet is the top-level container for applets so the components must be added to it. Similarly the frame is the top-level component for the standalone applications.

For more complicated GUI's it is convenient to divide the applets or frame into smaller regions. These regions might constitute, for example, a toolbar or a matrix of radio buttons. In java, GUI sub-regions are implemented most commonly with the panel container.

Panels, just like applets and frames can contain other components such as buttons, canvas, checkboxes, scrollbars, scrollpanes, text areas, textfields, and other panels.

Complicated GUI's sometimes have very complicated containment hierarchies of panels with in panels within panels. And so on, down through many layers of containment. As containers are also components so the methods available for components are also available for containers due to inheritance.

The most commonly used method for the containers is add() method using which the components are added the container its general form is.

void add (Component comp);

All the components are the sub-classes of the component class so any component can be added to the container using the add() method.

Layout Manager:

While adding components to a containers, one important issue is related to the position and size of the component. The components position can be specified in two ways.

(i) **Absolute positioning.**

(ii) **Relative positioning.**

In absolute positioning, we specify the exact location at which the component should be added. This method gives us fine control and may be the best choice if we design the GUI for just one platform.

For example, a component can be added in middle of container along horizontal direction, by specifying the co-ordinates of top-left corner of the component.

We can calculate these co-ordinates based on the container width and the component width. But the component will not be left in the middle. If the container width changes or the size of the component changes.

The other approach, relative positioning, may not allow us to specify the exact location of the components but may be suitable in the situation where we want that the component should always be in the middle of the container along the horizontal direction.

We can achieve this by specifying the relative position of the component like left, right, center etc. it is very common to center headings in documents using word processors. The headings remain centered even if the width of the page changes this is an example of relative positioning.

In java it is possible to use absolute positioning but it is not advisable. The AWT is designed to use relative positioning. We can specify component layouts with relative specifications, such as the component will be added to the right of the previous component, the component will be added below some other component, the component will appear at top, size of the component will be $\frac{1}{4}$ th of the container width etc. such specifications are useful even without knowledge of component sizes.

Java encourages relative positioning because it is platform independent and the same GUI should work on a wide variety of platforms without modifications.

Built-in java classes called layout managers handle the task of mapping relative positions to actual/physical positions. Java supports many types of layout managers, we will discuss about following layout managers.

1. FlowLayout

2. **GridLayout**
3. **BorderLayout**
4. **CardLayout**
5. **GridBagLayout**

Two Observations when working with layout managers:

- We do not have to bear the burden of specifying the exact position and dimension of each component.
- We no longer had the power to specify the exact position and dimensions of each component.

Why Java uses layout managers?

There are two reasons:

1. The theory lies in the position that precise layout (that is, specification in pixels of each component's size and position) is a repetitious and often performed task; therefore, according to OOP's layout functionality ought to be encapsulated into one or more classes to automate the task.

Certainly the layout managers eliminate a lot of development tedious. Many programmers dislike the idea of layout managers first, but come to appreciate them more and more as tedious choices are eliminated.

2. The practical reason for having layout manager stems from java's platform independence. In java, AWT components borrow their behaviour from the window system of the underlying hardware on which the JVM is running.

Thus on a Macintosh, an AWT buttons tools like any other Mac button; on a motif platform, a java button looks like any other motif button, and so on. The problem here is that buttons and other components have different sizes when instantiated on different platforms.

If java encouraged precise pixel-level sizing and positioning, there would be lot of java GUI's that looked exquisite on their platform of origin and terrible or even unusable, on other platforms.

As discussed above, layout managers solve the problem by using relative positioning.

Layout policy:

Every java component has preferred size. The preferred size expresses how big the component would like to be, barring conflict with a layout manager. Preferred size is generally the smallest size necessary to render the component in a visually meaningful way for example, a button's preferred size is the size of its label text, plus a little border of empty space around the text, plus the shadowed decorations that mark the boundary of the button thus a buttons preferred size is "just big enough". Preferred size is platform

dependent since component boundary decorations vary from system to system.

When a layout manager lays out its container's child components, it has to balance two considerations;

- (i) **layout policy and**
- (ii) **each component's preferred size**

First priority goes to enforcing layout policy. If honoring a component's preferred size would mean violating the layout policy, then the layout manager overrules the component's preferred size. Understanding a layout manager means understanding where it will place a component and also how it will treat a component's preferred size.

1. FlowLayout:

The functionality of the flow layout manager is encapsulated in the class `java.awt.FlowLayout`.

The Flow Layout Managers arranges components in horizontal rows. It is the default layout manager type for panels and applets.

The flow layout manager fits as many components as possible into the top row and moves the other component into second row. If no space is left in the second row, the component will move to the third row, and so on. This is quite similar to typing text in a document.

When we reach at the end of the current line, the next word moves to the next line, and so on. If we change the width or height of the document, the entire document is re-formatted.

The components always appear, left to right, in the order in which they were added to their container.

By default, the flow layout manager leaves a gap of five pixels between components in both the horizontal and vertical directions. This default can be changed by calling an overloaded version of the `FlowLayout` constructor, and passing in the desired horizontal and vertical gaps.

Within every row, the components are evenly spaced, and the cluster of components is centered. The alignment (sometimes called "justification") of the clustering can be controlled by passing a parameter to the `Flow Layout` constructor. The possible values are :

 **FlowLayout.LEFT**

🚩 FlowLayout.CENTER

🚩 FlowLayout.RIGHT

Example 17.1:

```
1.      import java.awt.*;
2.      class MyFrame1
3.      {
4.          public static void main(String args[ ])
5.          {
6.              Frame f=new Frame( );
7.              FlowLayout flow=new FlowLayout();
8.              f.setLayout(flow);
9.              Label l1 = new Label("Name : ");
10.             f.add(l1);
11.             TextField tf1=new TextField("Matrix");
12.             f.add(tf1);
13.             tf1.setBackground(Color.green);
14.             tf1.setForeground(Color.red);
15.             Button b1 = new Button("ok");
16.             Font f1 = new Font("Arial", Font.BOLD, 24);
17.             b1.setFont(f1);
18.             f.add(b1);
19.             f.setSize(500,100);
20.             f.setVisible(true);
21.         }
22.     }
```

Note:-If we reduce the size of the above window and three components do not fit in a single row then components are moved in the next row starting from last component.

Output:



Example 17.2

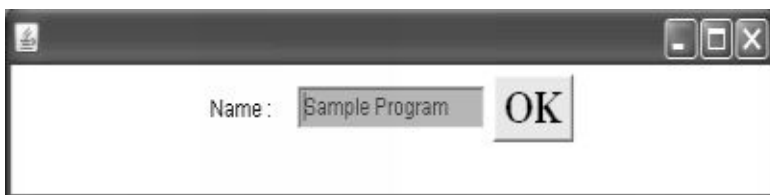
```
1.      import java.awt.*;
2.      public class MyFrame2 extends Frame
3.      {
4.          public static void main (String args [] )
5.          {
6.              MyFrame2 f=new MyFrame2();
7.      f.setLayout(new FlowLayout(FlowLayout.CENTER));
8.              System.out.println(f paramString());
9.              Label l1=new Label("Name :");
10.             f.add(l1);
11.             TextField tf1=new TextField("Sample Program");
12.             f.add(tf1);
13.             tf1.setBackground(Color.GREEN);
14.             tf1.setForeground(Color.RED);
15.             Button b1=new Button("OK");
16.             b1.setFont(new Font("Serif",Font.BOLD,24));
17.             //b1.setEnabled(false);
18.             Dimension d = b1.getSize( );
19.             System.out.println("Button Size :" + d.width + " " + d.height);
20.             f.add(b1);
21.             // f.setResizable(false);
22.             // f.setUndecorated(true);
23.             f.setVisible(true);
24.             Dimension d1=b1.getSize( );
25.             System.out.println("Button Size: " + d1.width + " " + d1.height);
26.             Dimension d2=f.getSize();
27.             System.out.println("Frame Size: " + d2.width + " " + d2.height);
28.             try
29.             {
30.                 Thread.sleep(5000);
```

```

31.         }
32.         catch(InterruptedException e) { }
33.         f.setVisible(false);
34.         f.setSize(500,100);
35.         f.setVisible(true);
36.         try
37.         {
38.             Thread.sleep(5000);
39.         }
40.         catch(InterruptedException e){ }
41.         f.setVisible(false);
42.         f.setBounds(20, 20, 100, 100);
43.         f.setVisible(true);
44.         Dimension d3=f.getSize();
45.         System.out.println("Frame Size : " + d3.width + " " + d3.height);
46.         System.out.println(f paramString());
47.     }
48. }

```

Output:



frame0,0,0,0x0,invalid,hidden,layout=java.awt.FlowLayout,title=,resizable,normal

Button Size :0 0

Button Size: 52 35

Frame Size: 123 34

Frame Size : 123 100

frame0,20,20,123x100,layout=java.awt.FlowLayout,title=,resizable,normal

Example 17.3

The components will be left aligned, if we change the line:

```
f.setLayout(new FlowLayout(FlowLayout.CENTER));
```

in the above program to

```
f.setLayout(new FlowLayout(FlowLayout.LEFT));
```

Output:



Example 17.4:

The components will be right aligned, if we change the line:

```
f.setLayout(new FlowLayout(FlowLayout.LEFT));
```

in the above program to

```
f.setLayout(new FlowLayout(FlowLayout.RIGHT));
```

Output:

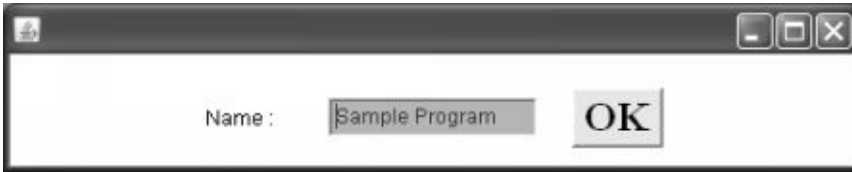


Example 17.5:-We can change the gap between components in a row (horizontal gap) and the gap between rows (vertical gap) by using the overloaded constructor while setting the layout:

```
f.setLayout(new FlowLayout(FlowLayout.CENTER, 20 ,20))
```

vgap(between rows) hgap(between components)

Output:



3. GridLayout:

The functionality of the Grid Layout Manager is encapsulated in the class `java.awt.GridLayout`. The Grid Layout Manager arranges components in tabular format in row and columns.

We can think of container space as a grid of rows and columns. We have to specify the number of rows and columns while specifying the layout.

The Grid Layout Manager fits as many components as the size of the row (number of columns specified in the layout) into the top row and moves the other component into second row, and so on. This is like the spreadsheet, which is divided into fixed number of rows and columns.

The flow layout manager always honors a component's preferred size. The GridLayout manager takes the opposite extreme: when it performs a layout in a given space, it ignores a component's preferred size.

Each row and column in a grid layout will be the same size, the overall area available to the layout is divided equally between the number of rows and between the number of columns. The grid layout uses "row major" notation, i.e. components appear in the order in which they were added, from left to right, row by row.

GridLayout manager behaves strangely when we add few components (that is, significantly fewer than the number of rows times the number of columns) or very many components (that is, more than the number of rows times the number of columns).

If the same components are to be laid in a taller, narrower frame, then every component is proportionally taller and narrower and vice-versa.

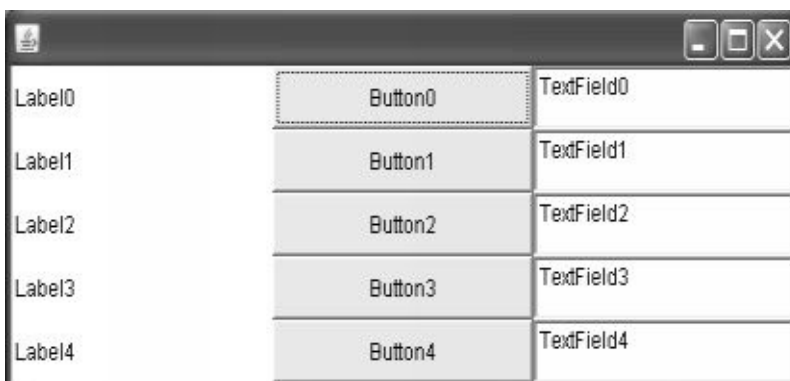
Example 17.6:

```

1. import java.awt.* ;
2. public class MyFrame6 extends Frame
3. {
4.     public static void main(String args[])
5.     {
6.         Frame f = new MyFrame6( );
7.         f.setLayout(new GridLayout(5,3));
8.         for(int row=0; row<5; row++)
9.         {
10.            f.add(new Label("Label" + row ));
11.            f.add(new Button("Button" + row ));
12.            f.add(new TextField("TextField" + row ));
13.        }
14.        f.setSize(500,200);
15.        f.setVisible(true);
16.    }
17. }

```

Output:



4. Border Layout:

The functionality of the BorderLayout manager is encapsulated in the class `java.awt.BorderLayout`. The BorderLayout manager is the default manager for frames, so sooner or later application programmers are certain to come to grips with it.

It enforces a very useful layout policy, but it is possibly less intuitive than either the flow or grid layout managers. The Flow layout manager always honor's a component's preferred size; the Grid layout manager never does. The Border layout manager does something in between.

The Border Layout Manager divides the available container space into five parts: North, South, East, West and Center. We can add one component to each region.

Each of the five regions may be empty or may contain one component (that is, no region is required to contain a component, but the regions can only contain one component).

The Border layout manager honors the preferred height of the North and South components, and forces them to be exactly as wide as the container. The North and South regions are useful for toolbars, status lines, and any other controls that ought to be as wide as possible, but no higher than necessary.

The East and West regions are the opposite of North and South. In East and West, a component gets its preferred width but has its height contained.

Here a component Extends vertically up to the bottom of the North component (if there is one) or to the top of the container (if there is no North component).

Similarly the component extends vertically down to the South component (if there is one) or to the bottom of the container (if there is no South component).

We can only put a single component in each region well, if that component is a container, then we can get multiple components displayed. The Border layout is not affected by the order in which we add components. Instead, we must specify which of the five regions will receive the component we are adding.

The overloaded version of add() takes two parameters:

- ✚ **First, the component being added, and**

- ✚ **Second, an object**

Proper use of Border layout manager requires that the second parameter be a constant defined in the Border Layout class itself. The five constants that we should know about are:

BorderLayout.NORTH

BorderLayout.SOUTH

BorderLayout.EAST

Border Layout.WEST

Border Layout.CENTER

The fifth region that a Border layout manager controls is called Center. Center is simply the part of a container that remains after North, South, East, and West have been allocated.

When adding a component to center, it is legal but very unwise, to emit the second parameter to add () call. In Java Platform, the Border layout manager will assume that We mean center;

However, in older versions, the behavior was unpredictable, and typically resulted in the component being entirely invisible.

Example 17.7

```
1.      import java.awt.*;
2.      public class MyFrame7
3.      {
4.          public static void main(String args[])
5.          {
6.              Frame f=new Frame( );
7.              Scrollbar sbRight=new Scrollbar(Scrollbar.VERTICAL);
8.              f.add(sbRight, BorderLayout.EAST);
9.              Scrollbar sbLeft=new Scrollbar(Scrollbar.VERTICAL);
10.             f.add(sbLeft, BorderLayout.WEST);
11.             Label labelTop=new Label("This is North");
12.             labelTop.setFont(new Font("Serif", Font.ITALIC, 36));
13.             labelTop.setForeground(Color.white);
14.             labelTop.setBackground(Color.black);
15.             f.add(labelTop, BorderLayout.NORTH);
16.             Label labelBottom=new Label("This is south");
```

```

17.         labelBottom.setFont(new Font("Monospaced", Font.BOLD, 18));
18.             labelBottom.setForeground(Color.white);
19.             labelBottom.setBackground(Color.black);
20.         f.add(labelBottom, BorderLayout.SOUTH);
21.         f.setSize(500,200);
22.         f.setVisible(true);
23.     }
24. }

```

Output:



Example 17.8:

```

1.     import java.awt.*;
2.     public class MyFrame8 extends Frame
3.     {
4.         public static void main(String args[])
5.         {
6.             Frame f=new MyFrame8( );
7.             //f.setLayout(new BorderLayout());
8.             f.add (new Button("N"), BorderLayout.NORTH);
9.             f.add (new Button("S"), BorderLayout.SOUTH);
10.            f.add (new Button("E"), BorderLayout.EAST);

```

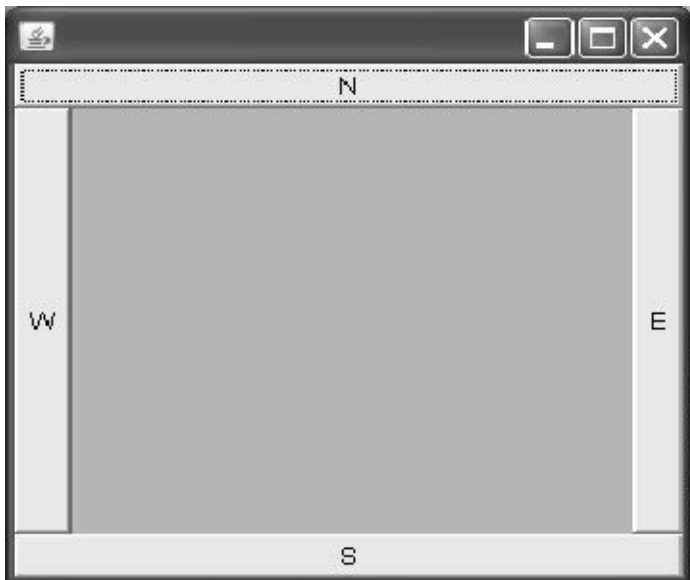


```

11.         f.add (new Button("W"), BorderLayout.WEST);
12.         Panel p=new Panel();
13.         p.setBackground(Color.green);
14.         f.add (p, BorderLayout.CENTER);
15.         //f.add (p); default is center
16.         f.setSize(300 ,300);
17.         f.setVisible(true);
18.     }
19. }

```

Output:



Example 17.9:

```

1.     import java.awt.*;
2.     class BorderLayout3 extends Frame
3.     {
4.         public static void main (String args[])
5.         {
6.             Frame f=new BorderLayout3();
7.             // f.setLayout(new BorderLayout());

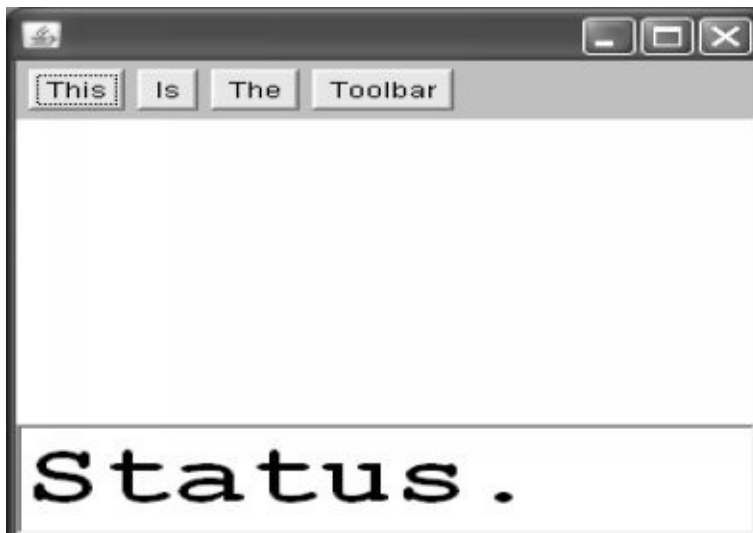
```

```

8.         Panel toolbar=new Panel();
9.         toolbar.setLayout(new FlowLayout(FlowLayout.LEFT));
10.        toolbar.setBackground(Color.lightGray);
11.        toolbar.add(new Button ("This"));
12.        toolbar.add(new Button ("Is"));
13.        toolbar.add(new Button ("The"));
14.        toolbar.add(new Button ("Toolbar"));
15.        f.add(toolbar, BorderLayout.NORTH);
16.        TextField status=new TextField("Status.");
17.        status.setFont(new Font("Monospaced",Font.BOLD,48));
18.        f.add (status, BorderLayout.SOUTH);
19.        f.setSize(300,300);
20.        f.setVisible(true);
21.    }
22. }

```

Output:



CardLayout:-

In CardLayout we can store many cards but one card will be shown at a time. Only one card will be visible at a time, but you can flip from one card to another. Here card means a Panel.

If we want to show two cards one at a time then we have to create first two

Panels one for each card and one main Panel for containing these two cards. This can be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input.

Constructors:

`CardLayout();`

creates a new card layout with gaps of size 0.

`CardLayout(int, int);`

Creates a new Card Layout with the specified horizontal and vertical gaps. The horizontal gaps are placed at the left and right edges. The vertical gaps are placed at the top and bottom edges.

From other layouts the `CardLayout` requires a bit more work. The cards are held typically in an object of type `panel`. When cards are added to the panel, they are usually given a name. Thus we will use the `add()` method when adding cards to a panel.

`void add(Component objpanel, Object objname)`

Here, `objname` is a string that specifies the name of the card whose panel is specified by `objpanel`.

Methods:

 **`void first(Container parent)`**

Goes to the first card of the container.

 **`void next(Container parent)`**

Goes to the next card of the given container. If the currently visible card is the last one, this method flips to the first card in the layout.

 **`void previous(Container parent)`**

Goes to the previous card of the given container. If the currently visible card is the first one, this method flips to the last card in the layout.

 **`void last(Container parent)`**

Goes to the last card of the container

 **`void show(Container parent, String name)`**

Goes to the component that was added to the layout with the given name. If no such component exists, then nothing happens.

The following steps are used to create a CardLayout

1. Creates a new `Panel` object and an object of `CardLayout`.
2. Set the layout for `Panel` object as `CardLayout` using its instance created in first steps.

3. Now for each card, creates a separate Panel object and add whatever components we want to add to that Panel like buttons, checkboxes, etc.

4. Each Panel object created in steps 3 is added to the main Panel object created in step 1, giving a unique card name for each card.

5. Finally the main panel is added to the applet window.

Example17.10:

```
1. import java.awt.*;
2. import java.awt.event.*;
3. import java.applet.*;
4. /*<applet code="CardLayoutTest" width=500 height=100></applet>*/
5. public class CardLayoutTest extends Applet implements ActionListener,
   MouseListener
6. {
7.     Panel mp;
8.     CardLayout c1;
9.     public void init()
10.    {
11.        Button b1=new Button("First");
12.        Button b2=new Button("Second");
13.        b1.addActionListener(this);
14.        b2.addActionListener(this);
15.        add(b1);
16.        add(b2);
18.        Panel p1=new Panel();
19.        Label l1=new Label("INDIA");
20.        p1.add(l1);
22.        Panel p2=new Panel();
23.        Label l2=new Label("USA");
24.        p2.add(l2);
26.        mp=new Panel();
27.        c1=new CardLayout();
28.        mp.setLayout(c1);
29.        mp.add(p1,"Country1");
```

```

30.         mp.add(p2,“Country2”);
31.         add(mp);
32.         addMouseListener(this);
33.     }
34.     public void actionPerformed(ActionEvent ae)
35.     {
36.         Button b=(Button)ae.getSource();
37.         if(b.getLabel().equals(“First”))
38.             c1.show(mp,“Country1”);
39.         else
40.             c1.show(mp,“Country2”);
41.     }
42.     public void mousePressed(MouseEvent me)
43.     {
44.         c1.next(mp);
45.     }
46.     public void mouseReleased(MouseEvent me){}
47.     public void mouseClicked(MouseEvent me){}
48.     public void mouseEntered(MouseEvent me){}
49.     public void mouseExited(MouseEvent me){}
50. }

```

Output:



GridBagLayout-

The GridBagLayout is the most complex and flexible of the standard layout managers. Although it sounds like it should be a subclass of GridLayout, but it is different entirely. With GridLayout, elements are arranged

Example17.11:

AWT components can be classified as:

- 1. Visual Components.**
- 2. Container Components.**
- 3. Menu Components.**

Visual Components and Container components both are part of the component hierarchy starting with the class java.awt.Component.

1. Methods of java.awt.Component class

Several methods are implemented by all the visual and container components, by virtue of inheritance from java.awt.component. (The menu components extend from java.awt.MenuComponent, So they do not inherit the same super class functionality). These methods are discussed below:

Dimension getSize()

Returns the size of this component in the form of a Dimension object, which has public data members height and width of type int.

void setBackground (Color c)

Sets the background color of a component. Generally background color is used for rendering the non- textual area of the component.

void setForeground (Color c)

Sets the foreground color of a component. Generally foreground color of a component is used for rendering text. If We do not explicitly set a component's foreground or background color, the component uses the foreground and background

color of its immediate container.

void setFont (Font f)

The setFont () method determines the font that a component will use for rendering any text that it needs to display. If We do not explicitly set a component's font, the component uses the font of its container.

void setEnabled(Boolean b)

If the argument is true, then the component has its normal appearance. If the argument is false, then the component is grayed out and does not respond to user input.

void setSize (Dimension d)

Sets the size of the component.

void setSize (int width, int height)

Sets the size of the component.

void setBounds (int x, int y, int width, int height)

Attempts to move and resize the component. The new location of the top- left corner is specified by x and y , and the new size is specified by width and height.

void setVisible (Boolean b)

This method takes a Boolean argument that dictates whether the component is to be seen on the screen. This method is generally used for frames.

Note-

if We have tried calling setSize () or setBounds () methods, we know that it is usually futile. The size and position that we attempt to give a component is overruled by a layout manager.

In fact, these two methods exist mostly for the use of layout managers. The major exception to this rule is the Frame class, which is not under the thumb of a layout manager and is perfectly willing to have we set its size or bounds.

Visual Components:

The visual components are the ones that users can actually see and interact with. The AWT supports the following visual components.

Button

Canvas

Checkbox

Choice

FileDialog

Label

List

ScrollPane

Scrollbar

TextArea

TextField

To use one of these components in a GUI, we first create an instance by calling the appropriate constructor. Then we add the component to a container.

Button:

The Button class, implements a push button.

Constructors

Button()

Button(String)

The constructor takes a string parameter that specifies the text of the button's label. When a button is pushed it sends an Action event.

Methods:

String

getLabel ()

Gets the label of this button.

void

setLabel (String label)

Sets the button's label to be the specified string.

void

addActionListener (ActionListener l)

Adds the specified action listener to receive action events from this button.

Example 17.12

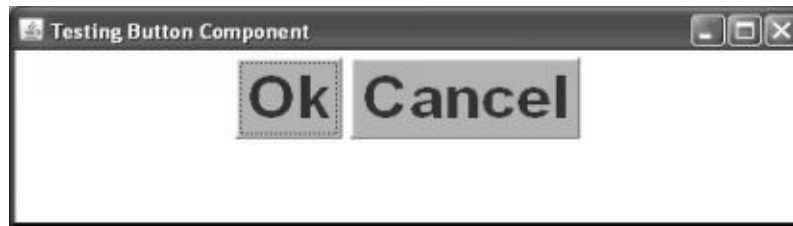
```
1.      import java.awt.*;
2.      class ButtonTest
3.      {
4.          public static void main(String args[])
5.          {
6.              Frame f=new Frame("Testing Button Component");
7.              f.setLayout(new FlowLayout());
8.              Font f1=new Font("Arial",Font.BOLD,40);
9.              Color c1=new Color(255,0,0);
10.             Color c2=new Color(0,255,0);
11.             Button b1 = new Button();
12.             Button b2 = new Button("Cancel");
13.             b1.setForeground(c1);
14.             b2.setForeground(c1);
15.             b1.setBackground(c2);
16.             b2.setBackground(c2);
17.             b1.setFont(f1);
18.             b2.setFont(f1);
19.             b1.setLabel("Ok");
20.             System.out.println(b2.getLabel());
```

```

21.         f.add(b1);
22.         f.add(b2);
23.         f.setSize(500,200);
24.         f.setVisible(true);
25.     }
26. }

```

Output:



Canvas:

A Canvas is a component, which has no default appearance or behavior. We can subclass canvas to create custom drawing regions, work areas, components and so on. Canvases receive input events from the mouse and the keyboard, it is up to the programmer to transform those inputs into a meaningful look and feel.

The default size (or, more properly, the preferred size) of a canvas is uselessly small. One way to deal with this problem is to use a layout manager that will resize the canvas. Another way is to call `setSize ()` on the canvas yourself, canvases are a rare case where this might actually work because the value we send into the `setSize ()` method becomes the canvas's preferred size.

Example 17.13

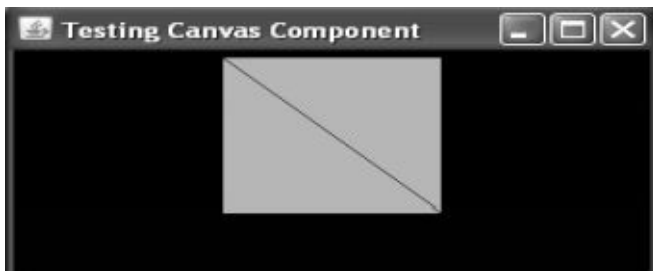
```

1.     import java.awt.*;
2.     class CanvasTest
3.     {
4.         public static void main (String args [])
5.         {
6.             Frame f = new Frame() ;
7.             f.setLayout(new FlowLayout());
8.             Canvas c=new MyCanvas();
9.             c.setSize(100, 100);
10.            f.add(c) ;

```

```
11.         f.setSize(300, 300 );
12.         f.setVisible(true);
13.     }
14.}
15.     class MyCanvas extends Canvas
16.{
17.         public void paint(Graphics g )
18.     {
19.         setBackground(Color.green);
20.         g.setColor(Color.red );
21.         g.drawLine(0, 0, 99, 99 );
22.     }
23.}
```

Output:



Checkbox:

A check box is a two state button. The two states are true (checked) and false (unchecked). The three basic forms of the checkbox constructors are:

Constructors-

Checkbox()

Creates a check box with an empty string for its label.

Checkbox (String label)

Creates a check box with the specified label.

Checkbox(String label, boolean state)

Creates a check box with the specified label and sets the specified state. If we do not specify an initial state, the default is false.

Methods:

String getLabel ()

Gets the label of this check box.

void setLabel (String label)

Sets this check box's label to be the string argument.

boolean getState ()

Determines whether this check box is in the "on" or "off" state.

void setState (boolean state)

Sets the state of this check box to the specified state.

void addItemListener (ItemListener)

Adds the specified item listener to receive item events from this check box.

Example 17.14

```
1.      import java.awt.*;
2.      class CheckboxTest
3.      {
4.          public static void main(String args[])
5.          {
6.              Frame f = new Frame("Testing Checkbox Component");
7.                  f.setLayout(new GridLayout(3,1));
8.                  Checkbox c1=new Checkbox();
9.                  Checkbox c2=new Checkbox("Lunch",true);
10.                 Checkbox c3=new Checkbox("Dinner");
```

```

11.         c1.setLabel("BreakFast");
12.         c3.setState(true);
13.         f.add(c1);f.add(c2);f.add(c3);
14.
15.         if(c1.getState())
16.             System.out.println(c1.getLabel());
17.         if(c2.getState())
18.             System.out.println(c2.getLabel());
19.         if(c3.getState())
20.             System.out.println(c3.getLabel());
21.
22.         f.setSize(100,100);
23.         f.setVisible(true);
24.     }}

```

Output:



Another Example-

```

1.     import java.awt.*;
2.     import java.awt.event.*;
3.     class CheckboxTest extends Frame implements ItemListener
4.     {
5.         Checkbox c1;
6.         TextField tf1;
7.         CheckboxTest()
8.         {
9.             setLayout(new GridLayout(2,1));
10.            c1=new Checkbox("Red");
11.            c1.addItemListener(this);

```

```

12.         add(c1);
13.         tf1=new TextField();
14.         add(tf1);
15.         setSize(100,100);
16.         setVisible(true);
17.     }
18.     public static void main(String args[])
19.     {
20.         new CheckboxTest();
21.     }
22.     public void itemStateChanged(ItemEvent ie)
23.     {
24.         if(c1.getState())
25.             tf1.setBackground(Color.red);
26.         else
27.             tf1.setBackground(Color.white);
28.     }
29. }

```

Radio button:

Checkboxes can be grouped together into checkbox groups, which have radio behavior. With radio behavior, only one member of a check box group can be true at any time; selecting a new member changes the state of the previously selected member to false. Many windows systems (Motif for example) implement radio groups as components in their own right.

We can implement radio behavior by creating multiple checkboxes and grouping them using java.awt. Checkbox Group Class. In Java the java.Awt.CheckboxGroup class is not a component; it is simply a non- visible class that organizes checkboxes. This means that Java imposes no restrictions on the spatial relationships among members of a check box group.

If we want to, we could put one member of a group in the upper left corner of a frame, another member in the lower-right corner, and a third member in a different frame altogether. Of course, the result would probably be contrary to both reason and most GUI style guides.


```

22.         f.setSize(100,100);
23.         f.setVisible(true);
24.     }
25. }

```

Output:



Choice:

A choice is a pull –down/drop-down list.

Methods:

void **add (String item)**
 Adds an item to this Choice menu.

void **addItemListener (ItemListener i)**
 Adds the specified item listener to receive item events from this choice menu.

String **getItem (int index)**
 Gets the String at the specified index in this Choice menu.

int **getItemCount ()**
 Returns the number of items in this Choice menu.

int **getSelectedIndex ()**
 Returns the index of the currently selected item.

String **getSelectedItem ()**
 Gets a representation of the current choice as a string.

void	insert (String item, int index)	Inserts the item into this choice at the specified position.
void	remove (int position)	Removes the first occurrence of item form the choice menu.
void	remove(String item)	Removes the first occurrence of item from the Choice menu.
void	removeAll ()	Removes all items from the choice menu.
void	select (int pos)	Sets the selected item in this choice menu to be the item at the specified position.
void	select (String Str)	Sets the selected item in this Choice menu to be the item whose name is equal to the specified string.

Example 17.16

```

1.      import java.awt.*;
2.      class ChoiceTest
3.      {
4.      public static void main(String args[]) throws InterruptedException
5.      {
6.          Frame f = new Frame("Testing Choice Component");
7.          f.setLayout(new FlowLayout());
8.          f.setSize(100,100);
9.          f.setVisible(true);
10.         Choice c1=new Choice();
11.         c1.add("BreakFast");
12.         c1.add("Lunch");
13.         c1.add("Dinner");
14.         f.add(c1);

```

```

15.         c1.select("Dinner"); // c1.select(2);
16.         Thread.sleep(1000);
17.         System.out.println(c1.getSelectedItem());
18.         System.out.println(c1.getSelectedIndex());
19.         System.out.println(c1.getItemCount());
20.         System.out.println(c1.getItem(0));
21.         c1.insert("Brunch",1);
22.         c1.remove(2); // c1.remove("Lunch");
23.     }
24. }

```

Output:



```

1.     import java.awt.*;
2.     import java.awt.event.*;
3.     class CheckboxItemTest extends Frame implements ItemListener
4.     {
5.         TextField tf1;
6.         Choice c1;
7.         CheckboxItemTest()
8.         {
9.             setLayout(new FlowLayout());
10.            c1=new Choice();
11.            c1.add("red");
12.            c1.add("green");
13.            c1.add("blue");
14.            tf1 = new TextField(8);

```

```

15.         tf1.setBackground(Color.red);
16.         c1.addItemListener(this);
17.         add(c1);
18.         add(tf1);
19.         setSize(200,200);
20.         setVisible(true);
21.     }
22.     public static void main(String args[])
23.     {
24.         new CheckboxItemTest();
25.     }
26.     public void itemStateChanged(ItemEvent ie)
27.     {
28.         String s=(String)ie.getItem();
29.         if(s.equals("red"))
30.             tf1.setBackground(Color.red);
31.         else if(s.equals("green"))
32.             tf1.setBackground(Color.green);
33.         else if(s.equals("blue"))
34.             tf1.setBackground(Color.blue);
35.     }
36. }

```

Label:

The simplest visible AWT component is label. Labels are not generally used to respond to user input, or to send out events.

Constructors:

Label ()

Constructs an empty label.

Label (String text)

Constructs a new label with the specified string of text, left justified.

Label (String text, int alignment)

Constructs a new label that present the specified string of text with the specified alignment.

The default alignment for labels is to the left. To set the alignment, use the third from of the constructor and pass in one of the following:

- 🚩 **Lable.LEFT**
- 🚩 **Label.CENTER**
- 🚩 **Label.RIGHT**

Methods:

String **getText ()**
Gets the text of the this label.

void **setText (String text)**
Sets the text for this label to the specified text.

Example 17.17

```
1.      import java.awt.*;
2.      class LabelTest
3.      {
4.          public static void main(String args[])
5.          {
6.              Frame f = new Frame("Testing Label Component");
7.              f.setLayout(new GridLayout(3,3));
8.              f.setFont(new Font("Arial",Font.PLAIN, 20));
9.              Label l1 = new Label("India",Label.LEFT);
```

```

10.         Label l2 = new Label("Pakistan",Label.LEFT);
11.         Label l3 = new Label("UK",Label.LEFT);
13.         Label l4 = new Label("India",Label.CENTER);
14.         Label l5 = new Label("Pakistan",Label.CENTER);
15.         Label l6 = new Label("UK",Label.CENTER);
17.         Label l7 = new Label("India",Label.RIGHT);
18.         Label l8 = new Label("Pakistan",Label.RIGHT);
19.         Label l9 = new Label("UK",Label.RIGHT);
21.         l3.setText("USA");
22.         System.out.println(l1.getText());
24.         f.add(l1);f.add(l4);f.add(l7);
25.         f.add(l2);f.add(l5);f.add(l8);
26.         f.add(l3);f.add(l6);f.add(l9);
28.         f.setSize(500,200);
29.         f.setVisible(true);
30.     }
31. }

```

Output:



List:

A list is a collection of text items, arranged vertically. If a list contains more items than it can display, it automatically acquires a vertical scroll bar.

Constructors:

List ()

Creates a new Scrolling list. This constructor does not specify number of visible

rows but produces a default preferred height of four rows. Of course, in many cases the actual height of a list will be dictated by a layout manager.

List (int rows)

Creates a new scrolling list initialized with the specified number of visible lines.

List (int rows, boolean multipleMode)

Creates a new scrolling list initialized to display to specified number of rows and allows multiple selections.

Example 17.18

```
1.      import java.awt.*;
2.      class ListTest
3.      {
4.          public static void main(String args[])
5.          {
6.              Frame f = new Frame ( ) ;
7.              f.setLayout (new FlowLayout ());
8.              List l1=new List(3,true);
9.              l1.add("India");
10.             l1.add("Nepal");
11.             l1.add("Shri Lanka");
12.             l1.add("USA");
13.             l1.add("UK");
14.             f.add(l1);
15.             System.out.println(l1.getRows());
16.             f.setSize(300, 300);
17.             f.setVisible(true);
18.         }
19.     }
```

Output:



The List has five items but only 3 visible rows, so a scroll bar is automatically provided to give access to the bottom two lines.

Note: Selecting an item in a list causes the list to send an Item event; double-clicking an item sends an Action Event.

void **add(String item)**

Add the specified item to the end of scrolling list.

void **add (String item, int index)**

Adds the specified item to the scrolling list at the position indicated by the index.

void **addActionListener (ActionListener i)**

Adds the specified item listener to receive action events form this list.

void **addItemListener (ItemListener i)**

Adds the specified item listener to receive item events form this list.

String **getItem (int index)**

Gets the item associated with the specified index.

int **getItemCount ()**

Gets the number of items in the list.

String[] **getItems()**

Gets the items in the list.

int **getRows ()**

Gets the number of visible lines in this list.

int **getSelectedIndex ()**

Gets the index of the selected item in the list.

int[] **getSelectedIndexes ()**

Gets the selected indexes in the list.

String **getSelectedItem ()**

Gets the Selected item in this scrolling list.

String[] **getSelectedItems ()**

Gets the selected items in this scrolling list.

void **remove (int position)**

Removes the item at the specified position from this scrolling list.

void **remove (String item)**

Removes the first occurrence of an item from the list.

void **removeAll ()**

Removes all items from this list.

void **replaceItem (String newValue, int index)**

Replaces the item at the specified index in the scrolling list with the new string.

void **select (int index)**

Selects the item at the specified index in the scrolling list.

void **setMultipleMode (boolean b)**

Sets the flag that determines whether this list allows multiple

selections.

```
1. import java.awt.*;
2. import java.awt.event.*;
3. class ListTest extends Frame implements ActionListener
4. {
5.     Button b1,b2,b3,b4,b5,b6;
6.     List l1,l2;
7.     TextField tf1,tf2;
8.     ListTest()
9.     {
10.         setLayout(new GridLayout(3,3));
11.         l1=new List(3,true);
12.         l2=new List(3,true);
13.         b1=new Button(">");
14.         b2=new Button(">>");
15.         b3=new Button("<");
16.         b4=new Button("<<");
17.         b5=new Button("Add");
18.         b6=new Button("Add");
19.         b1.addActionListener(this);
20.         b2.addActionListener(this);
21.         b3.addActionListener(this);
22.         b4.addActionListener(this);
23.         b5.addActionListener(this);
24.         b6.addActionListener(this);
25.         tf1=new TextField();
26.         tf2=new TextField();
27.         Panel p1=new Panel();
28.         p1.setLayout(new GridLayout(4,1));
29.         p1.add(b1);
30.         p1.add(b2);
```



```
63.         }
64.     else if(b==b3)
65.     {
66.         String s[]=l2.getSelectedItems();
67.         for(int i=0;i<s.length;i++)
68.         {
69.             l1.add(s[i]);
70.             l2.remove(s[i]);
71.         }
72.     }
73.     else if(b==b4)
74.     {
75.         String s[]=l2.getItems();
76.         for(int i=0;i<s.length;i++)
77.         {
78.             l1.add(s[i]);
79.             l2.remove(s[i]);
80.         }
81.     }
82.     else if(b==b5)
83.     {
84.         l1.add(tf1.getText());
85.     }
86.     else if(b==b6)
87.     {
88.         l2.add(tf2.getText());
89.     }
90.     }
91. }
```

TextField and TextArea:

The Text Field and Text Area classes implement one-dimensional and two-dimensional components for text input, display and editing. Both classes extend from the TextComponent super class.

Object Component TextComponent (TextField , TextArea)

Both classes (TextField and TextArea) have a variety of constructors, which offer the option of specifying or not specifying an initial string or preferred size. The constructors that do not specify a preferred size are most appropriate for situations where a layout manager will ignore the component's preferred size.

Constructors of Text Field Class:

TextField()

Constructs a new text field.

TextField(int columns)

Constructs a new empty text field with the specified number of columns.

TextField(String text)

Constructs a new text field initialized with the specified text.

TextField(String text, int columns)

Constructs a new text field initialized with the specified text to be displayed, and wide enough to hold the specified number of columns.

Constructors of Text Area Class:

TextArea ()

Constructs a new text area with the empty string text.

TextArea (int rows, int columns)

Constructs a new text area with the specified number of row & columns and the empty string as text.

TextArea (String text)

Constructs a new text area with the specified text.

TextArea (String text, int rows, int columns)

Constructs a new text area with the specified text, and with the specified number of rows and columns.

TextArea (String text, int rows, int columns, int ScrollbarPolicy)

Constructs a new text area with the specified text, and with the rows, columns, and scroll bar policy as specified.

The text Area class defines several constants that can be supplied as values for the scrollbar Policy argument.

SCROLLBARS_BOTH,

SCROLLBARS_VERTICAL_ONLY,

SCROLLBARS_HORIZONTAL_ONLY,

SCROLLBARS_NONE.

For both classes, there are some surprising issues to the number of columns parameter.

First, the number of columns is a measure of width in terms of columns of text, as rendered in a particular font. A 25-column text area with a tiny font will be very narrow, while a 5-column text area with a huge font will be extremely wide.

Next, there is the problem of proportional fonts. For a fixed width font, it is obvious what the column width should be. For proportional font, the column width is taken to be the average of all the font's character widths. This average is a simple average of all the characters in the set; it does not take into account frequency of characters use. So in most cases, text components that contain largely lowercase letters will display more than the requested number of characters.

Methods in text Component Class:

String **getSelectText ()**

Returns the selected text from the text that is presented by this text

component.

String **getText ()**

Returns the text that is presented by this text component.

void **SetEditable(Boolean b)**

Sets the flag that determines whether or not this text component is editable.

void **setText Srting t)**

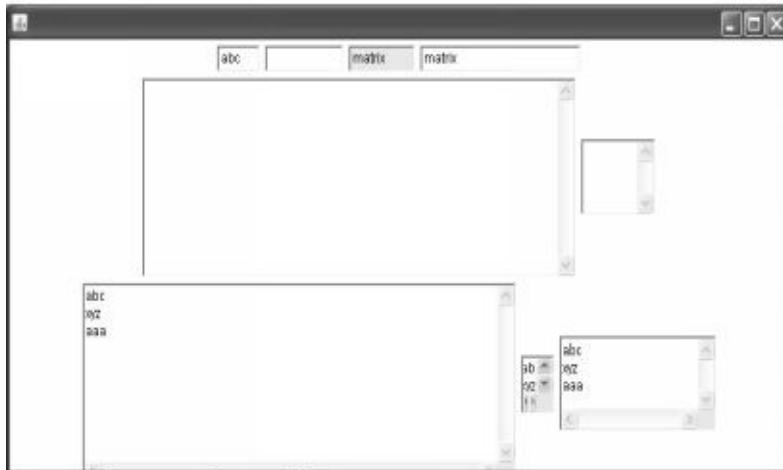
Sets the text that is presented by this text component to be the specified text.

Example 17.19

```
import java.awt.*;
1.     class TextFieldTest
2.     {
3.         public static void main(String args[])
4.         {
5.             Frame f=new Frame() ;
6.             f.setLayout(new FlowLayout ());
7.             TextField tf1=new TextField();
8.             TextField tf2=new TextField(8);
9.             TextField tf3=new TextField("matrix");
10.            TextField tf4=new TextField("matrix",20);
11.            TextArea ta1=new TextArea();
12.            TextArea ta2=new TextArea(3,8);
13.            TextArea ta3=new TextArea("abc\nxyz\naaa");
14.            TextArea ta4=new TextArea("abc\nxyz\naaa",2,2);
15.            TextArea ta5=new TextArea("abc\nxyz\naaa",4,20,
                TextArea.SCROLLBARS_BOTH);
16.            f.add(tf1);f.add(tf2);f.add(tf3);f.add(tf4);
17.            f.add(ta1);f.add(ta2);f.add(ta3);f.add(ta4);
```

```
18.         f.add(ta5);
19.         tf1.setText("abc");
20.         System.out.println(tf3.getText());
21.         tf3.setEditable(false);
22.         f.setSize(800, 400);
23.         f.setVisible(true);
24.     }
25. }
```

Output:



FileDialog:

This class represents a file open or file save dialog. A file dialog is a modal dialog, which means input from the dialog's parent frame will be directed exclusively to the dialog, as long as dialog remains visible on the screen. The dialog is automatically removed when the user specifies a file or clicks the cancel button. The most useful File dialog constructor has the following form:

FileDialog(Frame parent, String title, int mode)

The dialog's parent is the frame over which the dialog will appear. The title string appears in the dialog's title bar.

The mode should be either.

- **FileDialog.LOAD** or
- **FileDialog.SAVE**

After the user has specified a file, the name of the file or its directory can be retrieved with the following methods.:

String **getDirectory ()**

Gets the selected directory of this file dialog.

String **getFile ()** Gets the selected file of this file dialog.

Example 17.20

```

1.      import java.awt.*;
2.      public class FileDialogTest
3.      {
4.          public static void main(String args[])
5.          {
6.              Frame f=new Frame( );
7.              f.setLayout(new FlowLayout( ));
8.              f.setSize(200, 200);
9.              f.setVisible(true);
10.         FileDialog fd=new FileDialog (f, "Load file", FileDialog.LOAD);
11.         fd.setVisible(true);
12.         System.out.println(fd.getFile());
13.         System.out.println(fd.getDirectory());
14.     }
15. }
```

ScrollPane:

A scrollPane can contain a single component, which may be taller or wider than the scroll pane itself.

If the container component is larger than the scroll pane, then the default behavior of the scroll pane is to acquire horizontal and/ or vertical scroll bars as

needed.

There are two constructors for this class:

ScrollPane()

Create a new scroll pane container with a scrollbar display policy of “as needed”.

ScrollPane(int scrollbarDisplayPolicy)

Create a new scrollpane container with the specified scroll bar behavior. The scroll pane display policy should be one of:

`ScrollPane.SCROLLBARS_AS_NEEDED` (default)

`ScrollPane.SCROLLBARS_ALWAYS`

`ScrollPane.SCROLLBARS_NEVER`

The Code listed below creates a scroll pane with default (As_NEEDED) scroll bar behavior. The scroll pane contains a very large button; So the scroll bars will definitely be needed.

Example 17.21

```
1.      import java.awt.*;
2.      class SPaneTest
3.      {
4.          public static void main(String args[])
5.          {
6.              Frame f=new Frame();
7.              f.setLayout(new GridLayout(1,2));
8.              f.setSize(250, 150);
9.              ScrollPane spane=new ScrollPane();
10.             spane.setBackground(Color.green);
11.             f.add(spane);
```

```

12.          f.add(new TextArea());
13.          Button b1=new Button("Push Me !");
14.          b1.setFont(new Font("Serif", Font.ITALIC, 100));
15.          spane.add(b1);
16.          f.setVisible(true);
17.          }
18.      }

```

Output:



Scrollbar-

The Scrollbar component that adjusts lists and scroll panes is available as a component in its own right. There are three constructors:

Scrollbar ()

Constructs a new vertical scroll bar.

Scrollbar (int orientation)

Constructs a new scroll bar with the specified orientation.

Scrollbar (int orientation, int initialValue, int sliderSize, int minValue, int maxValue)

Constructs a new scroll bar with the specified orientation initial value, slider size, and minimum and maximum values. The sliderSize parameter controls the size of the slider, but not in pixel units.

For constructors that take an orientation parameter, this value should be one of:

- ✚ Scrollbar.HORIZONTAL
- ✚ Scrollbar.VERTICAL

Example 17.22

```
1.      import java.applet.*;
2.      import java.awt.*;
3.      import java.awt.event.*;
4.      /*<applet code = "RGB" width=500 height=200></applet> */
5.      public class RGB extends Applet implements AdjustmentListener
6.      {
7.          TextField tf1;
8.          Scrollbar sb1,sb2,sb3;
9.          public void init()
10.         {
11.             setLayout(new GridLayout(4,1));
12.             sb1 = new Scrollbar(Scrollbar.HORIZONTAL,0,10,0,255);
13.             sb2 = new Scrollbar(Scrollbar.HORIZONTAL,0,10,0,255);
14.             sb3 = new Scrollbar(Scrollbar.HORIZONTAL,0,10,0,255);
15.             sb1.addAdjustmentListener(this);
16.             sb2.addAdjustmentListener(this);
17.             sb3.addAdjustmentListener(this);
18.             tf1 = new TextField(8);
19.             add(tf1);add(sb1);
20.             add(sb2);add(sb3);
21.         }
22.         public void adjustmentValueChanged(AdjustmentEvent ae)
23.         {
24.             int a=0,b=0,c=0;
25.             a = sb1.getValue();
26.             b = sb2.getValue();
27.             c = sb3.getValue();
28.             Color c1 = new Color(a,b,c);
29.             tf1.setBackground(c1);
30.         }
```



CHAPTER

∞ 18 ∞

(INTRODUCTION TO AWT EVENTS)

Introduction-

When a normal Java program (which is not GUI based) needs input, it prompts the user and then calls some input method, such as `readline()` i.e. the interaction is initiated by the program. This is not the way in which gui-based programs behave. The user initiates interaction with the program rather than program initiating the action.

For example , in a word processing software, user initiates action by clicking on different buttons, which generates an event and some piece of code is executed as a result and accordingly some action takes place. The clicking on save button will invoke the method which saves the contents to a file.

Interface	Event(Methods)	class	Methods
ActionListener	actionPerformed()	ActionEvent ae	getSource
ItemListener	itemStateChanged()	ItemEvent ie	getItem
AdjustmentListener	adjustmentValueChanged()	AdjustmentEvent ae	getAdjustable()
MouseListener	mouseEntered	MouseEvent me	getX()
MouseAdapter	mouseExited		getY()
	mouseClicked		getPoint()
	mousePressed		getClickCount()
	mouseReleased		getButton()
			getModifiersEx()
			getModifiersExText()
MouseMotionListener	mouseMoved	MouseEvent me	same as prev.
MouseMotionAdapter	mouseDragged		
KeyListener	keyTyped	KeyEvent ke	getKeyChar()
KeyAdapter	keyPressed		getKeyCode()
	keyReleased		
FocusListener	focusGained	FocusEvent fe	getSource()
FocusAdapter	focusLost		
WindowListener	windowOpen	WindowEvent we	getWindow()
WindowAdapter	windowClosing		
	windowClosed		
	windowActivated		
	windowDeactivated		
	windowIconified		
	windowDeiconified		

Java uses event delegation model for event handling. In the event delegation model a component may be told which object or should be notified when the component generates a particular kind of event.

If a component is not interested in an event type, then events of that type will not be propagated. The delegation model is based on the following key concepts:

- 🚩 Event classes.

- ✚ Event listeners.

- ✚ Adapters.

Event classes:

Clicking on a button results in an event. The event has lot of information associated with it, like: time, component generating the event, co-ordinates of the point at which clicking took place etc.

This information is put inside an event class. The concept is similar to exception handling. Whenever an exception occurs an object of appropriate exception class is created and all the exception related information is put into this object.

We have hierarchy of event classes, which is some what similar to the hierarchy of exception classes. Whenever some event occurs, it is detected by event handling, mechanism, which creates objects of appropriate event class and puts all the event related information into it.

Event listeners:

Event listeners are responsible for taking appropriate action when an event occurs. An event listener is an object to which a component has delegated the task of handling a particular kind of event .

When the component experiences input, an event of the appropriate type is constructed, the event is then passed as the parameter to a method call on the listener. A listener must implement the interface that contains that event handling method.

We can compare the event listeners with the catch blocks for handling the exceptions. We need to provide a catch block corresponding to the exception to be handled. Similarly if we want to handle an event generated by some component, we need to provide appropriate listener.

The listeners are associated with the components. Different events have different listeners. When event occurs, an appropriate listener method is called and event object is passed as parameter.

Adapters:

The listeners defined in the language are interfaces only. We need to provide implementation of the appropriate listener. Some listeners have 2 or more methods.

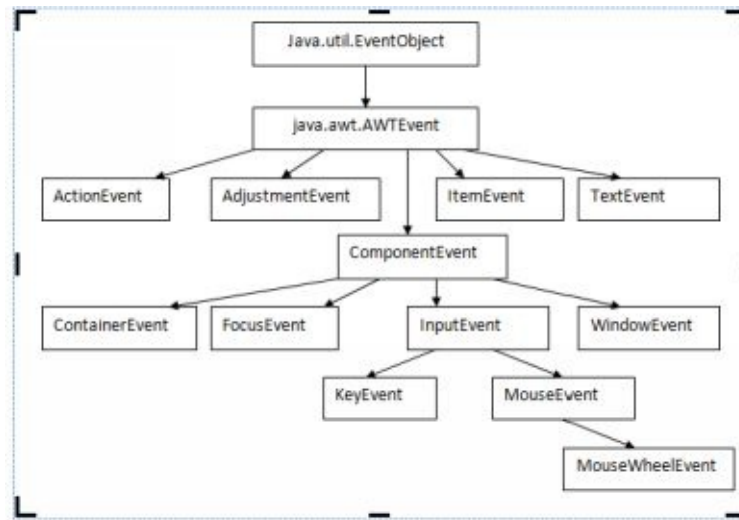
To implement a listener having more than one method, we need to provide implementation of all the methods even if we are interested only in one method. the adapter classes provide the solution to this problem.

An adapter class provides dummy implementation of all the methods of a listener. This helps in reducing programming efforts. If we want to use just one method of a listener class containing say 5 methods then we can simply extent the adapter class and override the desired method.

Event hierarchy:

The AWT has a well-defined event hierarchy. Normally the classes corresponding to leaf nodes in the event hierarchy represent the actual events. While non-leaf classes act as base classes and encapsulate .

AWT event hierarchy is described in the following diagram.



The `java.util.eventObject` class:

This is the top most super class of all the event classes. It is a very general class, with only one method of interest, which returns the object that originated the event.

Object `getSource()`

The `java.awt.AWTEvent` Class:

One sub class of event object is `java.awt.AWTEvent` , which is the super class of all the delegation model event classes. Again, there is only one method of interest, which return the id of the event.

`int getID()`

An event's ID is an int that specifies the exact nature of the event. For example, an instance of the mouseevent class can represent one of seven occurrences: click , Drag, Entrance, Move, Exit, Press, Release. Each of these possibilities is represented by an int `MouseEvent` . `MouseEvent.MOUSE_CLICKED`, `MouseEvent.MOUSE_DRAGGED`, and so on.

The sub classes of `java.awt.AWTEvent` represent the various event types that can be generated by the various AWT components, and contain all necessary information relating to the activity that triggered the event.

The non-super class event types(i.e those that are actually fired by component) are:

1. **ActionEvent:** general by activation of components for example .it is generated on button click, on double-click on list item and on pressing enter in a TextField.
2. **AdjustmentEvent:** generated by adjustment of adjustable components such as scroll bars.
3. **ContainerEvent:** generated when components are added to or removed from a container.

4. **FocusEvent**: generated when a component receives or loses input focus.
5. **ItemEvent**: generated when an item is selected from a list, choice, or checkbox.
6. **KeyEvent** : generated by keyboard activity.
7. **MouseEvent**: generated by mouse activity.
8. **MouseWheelEvent**: An event, which indicates that the mouse wheel was rotated in a component.
9. **WindowEvent**: generated by window activity (such as iconifying or deiconifying.)
10. **TextEvent**: generated when a text component is modified.

The java.awt.event package:

This package provides the interfaces and classes for dealing with different types of events fired by AWT components. The events are fired by event sources. An event listener registers with an event source to receive notifications about the events of a particular type.

This package defines events and event listeners, as well as event listener adapters, which are convenience classes to make easier the process of writing event listeners.

Event Listener Interfaces:

Each event class has a corresponding listener interface and defines one or more methods. The most commonly used listener interfaces are described in the following sections.

The Action Listener Interface:

This is the listener interface for receiving action events. The class that is interested in processing an action event implement this interface, and the object created with that class is registered with a component, using the component's `addActionListener()` method. When the action event occurs, the action listener's `actionPerformed()` method is invoked whose signatures are:

void	actionPerfomed (ActionEvent e) Invoked when an action occurs.
------	---

The AdjustmentListerner Interface:

This is the listener interface for receiving adjustment events.

void	adjustmentValueChanged (Adjustment event e) Invoked when the value of the thumb/slider changes.
------	---

The ContainerListener interface:

This is the listener interface for receiving events when a component is added to or removed from a container.

Methods:

void	componentAdded (containerEvent e)	Invoked when a component is added to the container.
void	componentRemoved (containerEvent e)	Invoked when a component is removed from the container.

The FocusListener Interface:

This is the listener interface for receiving events when a component gains or loses focus.

Methods:

void	focusGained(focusEvent e)	Invoked when a component gains the keyboard focus.
void	focusLost (focusEvent e)	Invoked when a component loses the keyboard focus.

The ItemListener Interface:

This is the listener interface for receiving item events.

Method:

void *itemStateChanged (ItemEvent e)*
 Invoked when an item has been selected or deselected by the user.

The keyListener Interface:

This is the listener interface for receiving keyboard events(keystrocks)

void	keyPressed(KeyEvent e)	Invoked when a key is pressed.
void	key Released (KeyEvent e)	Invoked when a key is released.

void	keyTyped (KeyEvent e) Invoked when a key is typed (pressed and released.)
------	---

The MouseListener Interface:

This is the listener interface for receiving mouse events(press , release. Click, enter, and exit) on a component.

Method:

void **mouseClicked(MouseEvent e)**
Invoked when the mouse button is clicked (pressed and released) on a Component.

void **mouseEntered(MouseEvent e)**
Invoked when the mouse enters a component.

void **mouseExited(MouseEvent e)**
Invoked when a mouse exits a component.

void **mousePressed(MouseEvent e)**
Invoked when a mouse button is pressed on a component.

void **mouseReleased(MouseEvent e)**
Invoked when a mouse button is released on a component.

MouseMotionListener:

This is the listener interface for receiving mouse motion events (move and drag) on a component.

Method:

void **mouseDragged(MouseEvent e)**
Invoked when a mouse button is pressed on a component and then dragged.

void **mouseMoved(MouseEvent e)**
Invoked when a mouse cursor is moved over a component but no buttons is pushed.

The MouseWheelListener Interface:

This is the listener interface for receiving mouse wheel events on a component.

Method:

void **mouseWheelMoved(MouseEvent e)**
Invoked when the mouse wheel is rotated.

The WindowListener Interface:

when the window's status changes by virtue of being opened, closed, activated or deactivated, iconified or deiconified, the relevant method in the listener object is invoked, and the windowevent is passed to it.

Methods:

void windowActivated(WindowEvent e)

Invoked when the window is set to be the active window.

void windowClosed(WindowEvent e)

Invoked when the window has been close as the result of calling Dispose on the window.

void windowClosing(WindowEvent e)

Invoked when the user attempts to close the window from the window's system menu.

void windowDeactivated(WindowEvent e)

Invoked when a window is no longer the active window.

void windowDeiconified(WindowEvent e)

invoked when a window is changed from a minimized to a normal state.

void windowIconified(WindowEvent e)

Invoked when a window is changed from a normal to a minimized state.

void windowOpened (WindowEvent e)

Invoked the first time a window is made visible.

Handling ActionEvent:

Example18.1

The following example explains how to handle the actionevent generated as a result of clicking on a button.

```
1.      import java.awt.*;
2.      import java.awt.event.*;
3.      class A1
4.      {
5.          public static void main(String args[])
6.          {
7.              Frame f=new Frame();
```

```

8.         f.setLayout(new FlowLayout());
9.         f.setSize(500,100);
10.        Button b1=new Button("Push me");
11.        Button b2=new Button("Click me");
12.        b1.addActionListener(new B());
13.        b2.addActionListener(new C());
14.        f.add(b1);
15.        f.add(b2);
16.        f.setVisible(true);
17.    }
18. }
19. class B implements ActionListener
20. {
21.     public void actionPerformed(ActionEvent ae)
22.     {
23.         System.out.println("Button Pushed");
24.     }
25. }
26. class C implements ActionListener
27. {
28.     public void actionPerformed(ActionEvent ae)
29.     {
30.         System.out.println("Button Clicked");
31.     }
32. }

```

Output:



Example 18.2:

```

1.     import java.awt.*;
2.     import java.awt.event.*;

```

```

3.     class A2 implements ActionListener
4.     {
5.         static Button b1=null,b2=null;
6.         public static void main(String args[])
7.             {
8.                 Frame f=new Frame();
9.                 f.setLayout(new FlowLayout());
10.                f.setSize(500,100);
11.                b1=new Button("Push me");
12.                b2=new Button("Click me");
13.                b1.addActionListener(new A2());
14.                b2.addActionListener(new A2());
15.                f.add(b1);
16.                f.add(b2);
17.                f.setVisible(true);
18.            }
19.            public void actionPerformed(ActionEvent ae)
20.            {
21.                if(ae.getSource() == b1)
22.                    System.out.println("Button Pushed");
23.                else
24.                    System.out.println("Button Clicked");
25.            }
26.        }

```

Output: Same as Previous Example.

Example 18.3:

```

1.     import java.awt.*;
2.     import java.awt.event.*;
3.     class A3 implements ActionListener
4.     {
5.         Button b1=null,b2=null;
6.         A3()

```

```

7.         {
8.         Frame f=new Frame();
9.             f.setLayout(new FlowLayout());
10.            f.setSize(500,100);
11.            b1=new Button("Push me");
12.            b2=new Button("Click me");
13.            b1.addActionListener(this);
14.            b2.addActionListener(this);
15.            f.add(b1);
16.        f.add(b2);
17.            f.setVisible(true);
18.        }
19.        public static void main(String args[])
20.        {
21.            new A3();
22.        }
23.        public void actionPerformed(ActionEvent ae)
24.        {
25.            if(ae.getSource() == b1)
26.                System.out.println("Button Pushed");
27.            else
28.                System.out.println("Button Clicked");
29.        }
30.    }

```

Output: Same as Previous Example.

Example 18.4

```

1.        import java.awt.*;
2.        import java.awt.event.*;
3.        class A4 implements ActionListener
4.        {
5.            static Button b1=null,b2=null;
6.            static TextField tf1=null,tf2=null, tf3=null;

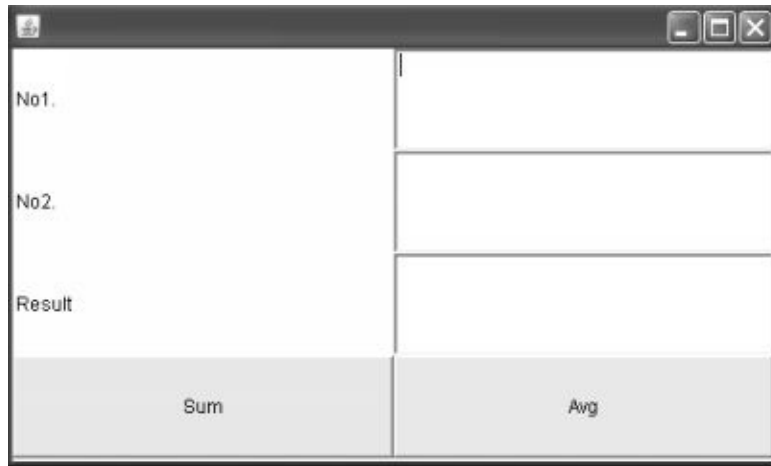
```

```

7.      public static void main(String args[])
8.          {
9.              Frame f=new Frame();
10.                 f.setLayout(new GridLayout(4,2));
11.                 f.setSize(500,300);
12.                 Label l1 = new Label("No1.");
13.                 Label l2 = new Label("No2.");
14.                 Label l3 = new Label("Result");
15.                 tf1 = new TextField();
16.                 tf2 = new TextField();
17.                 tf3 = new TextField();
18.                 b1=new Button("Sum");
19.                 b2=new Button("Avg");
20.                 b1.addActionListener(new A4());
21.                 b2.addActionListener(new A4());
22.                 f.add(l1);f.add(tf1);
23.                 f.add(l2);f.add(tf2);
24.                 f.add(l3);f.add(tf3);
25.                 f.add(b1);f.add(b2);
26.                 f.setVisible(true);
27.          }
28.      public void actionPerformed(ActionEvent ae)
29.          {
30.              int a,b,c;
31.              float av;
32.              a=Integer.parseInt(tf1.getText());
33.              b=Integer.parseInt(tf2.getText());
34.              if(ae.getSource() == b1)
35.                  tf3.setText(" " +(a+b));
36.              else
37.                  tf3.setText(" " + (a+b)/2.0f);
38.          } }

```

Output:



Note: a component may have multiple listeners associated with it . there is no guarantee that listeners will be notified in the order in which they were added.

There is also no guarantee that all listener notification will occur in the same thread, thus, listeners must take precautions against corrupting shared data.

An event listener may be removed from a component's list of listeners by calling `remove XXXListener()` method, passing in the listener to be removed.

For example ,code below removes action listener `al` from button `btn`:

```
btn.removeActionListener(al);
```

Handling window events

Example 18.5:

```
1.      import java.awt.*;
2.      import java.awt.event.*;
3.      class A5
4.      {
5.          public static void main(String args[])
6.          {
7.              Frame f= new Frame();
8.              f.setLayout(new FlowLayout());
9.              f.setSize(300,300);
10.             f.addWindowListener(new B( ));
11.             f.setVisible(true);
12.         }
13.     }
14.     class B implements WindowListener
```



```
15.     {
16.         public void windowOpened (WindowEvent we)
17.         {
18.             System.out.println ("Window Opened");
19.         }
20.         public void windowActivated(WindowEvent we)
21.         {
22.             System.out.println ("Window Activated");
23.         }
24.         public void windowDeactivated(WindowEvent we)
25.         {
26.             System.out.println ("window Deactivated");
27.         }
28.         public void windowClosing(WindowEvent we)
29.         {
30.             System.out.println("Window Closing");
31.             Frame f = (Frame) we.getWindow( );
32.             f.dispose();
33.         }
34.         public void windowClosed(WindowEvent we)
35.         {
36.             System.out.println("Window Closed");
37.         }
38.         public void windowIconified(WindowEvent we)
39.         {
40.             System.out.println("Window Iconified");
41.         }
42.         public void windowDeiconified(WindowEvent we)
43.         {
44.             System.out.println("Window Deiconified");
45.         }
46.     }
```

Adapters:

Some of the event listener interfaces have several methods. The largest interface, WindowListener has seven methods. Suppose we want to catch window Closing () event, so that the code for closing window can be executed. Although we need only one method window Closing () in the class implementing Window Listener but we will have to provide dummy implementation for all the methods.

The java.awt.event package provides several adapter classes, one for each listener interface that defines more than just a single method.

An adapter is simply a class that implements an interface by providing do nothing methods. For example, the Window Adapter class implements the Window Listener interface with seven do- nothing methods.

Adapter classes and Their Interfaces:

Adapter Class	Listener Interfaces
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

Example 18.6 Use of the WindowAdapter Class.

```
1.      import java.awt.*;
2.      import java.awt.event.*;
3.      class A6
4.      {
5.          public static void main (String args[])
6.          {
7.              Frame f= new Frame();
8.              f.setLayout(new FlowLayout());
9.              f.setSize(300,300);
10.             f.addWindowListener(new B( ));
11.             f.setVisible(true);
12.         }
13.     }
14.     class B extends WindowAdapter
```

```

15.      {
16.          public void windowClosing(WindowEvent we)
17.          {
18.              System.out.println("Window Closing");
19.              Frame f = (Frame) we.getWindow ( );
20.              f.dispose();
21.          }
22.      }

```

The InputEvent class:

This is the root event class for all component level input events. Input events are delivered to listeners before they are processed normally by the source where they originated. This allows listeners and component subclasses to “consume” the event so that the source will not process them in their default manner. For example, consuming mouse pressed events on a Button component will prevent the Button from being activated.

Methods:

void **consume()**

Consumes this event so that it will not be processed in the default manner by the source that originated it.

int **getModifiers ()**

Returns the Modifier mask for this event.

int **getModifiersEx ()**

Returns the extended modifier mask for this event.

static String **getModifiersExText (int modifiers)**

Returns a String describing the extended modifier Keys such as “shift”, “Ctrl + Shift” etc.

long **getWhen ()**

Returns a timestamp of when this event occurred.

boolean **isAltDown ()**

Returns whether or not the Alt modifier is down on this event.

boolean **isConsumed ()**

Returns whether or not this event has been consumed.

boolean **isControlDown ()**

Returns whether or not the Control modifier is down on this event.

boolean **isShiftDown ()**

Returns whether or not the shift modifier is down on this event.

The Mouse Event Class:

This is sub- class of Input Event class. This event indicates that a mouse action occurred in a component. A mouse action is considered to occur in a particular component if and only if the mouse cursor is over the un-obscured part of the component's children or by a menu or by a top- level widow.

This event is used both for Mouse events (click, enter, press, exit, release) and Mouse motion events (move and drag).

Methods:

int **getButton()**
Returns which, if any, of the mouse buttons has changed state.

int **getClickCount()**
Returns the number of mouse clicks associated with this event.

static String **getMouseModifiersText (int modifiers)**
Returns a String describing the modifier Keys that were down during the event, such as "Shift", or "Ctrl + shift".

point **getPoint ()**
Returns the x, y position of the event relative to the source component.

int **getX ()**
Returns the horizontal x position of the event relative to the source component.

int **getY ()**
Returns the vertical y position of the event relative to the source component.

boolean **isPopupTrigger ()**
Returns whether or not this mouse event is the popup menu trigger event for the platform.

Note: Popup menus are triggered differently on different systems. Therefore, isPopupTrigger () should be checked in both mouse Pressed and mouse Released for proper cross-platform functionality.

Handling Mouse Events:

Example 18.7

```
1.     import java.awt.*;
2.     import java.awt.event.*;
3.     class A7
4.     {
5.         public static void main(String args[])
6.         {
7.             Frame f= new Frame();
8.             f.setLayout(new FlowLayout());
9.             f.setSize(300,300);
10.            Button b1 = new Button("Push Me");
11.            b1.addMouseListener(new B());
12.            b1.addMouseMotionListener(new C());
13.            b1.addActionListener(new D());
14.            f.add(b1);
15.            f.setVisible(true);
16.        }
17.    }
18.    class B implements MouseListener
19.    {
20.        public void mouseEntered(MouseEvent me)
21.        {
22.            System.out.println("Mouse Entered...");
23.            System.out.println(me.getX());
24.            System.out.println(me.getY());
25.        }
26.        public void mouseExited(MouseEvent me)
27.        {
28.            System.out.println("Mouse Exited...");
29.            Point p1=me.getPoint();
30.            System.out.println(p1.x);
31.            System.out.println(p1.y);
32.        }
```

```
33.         public void mousePressed(MouseEvent me)
34.         {
35.             System.out.println("Mouse pressed...");
36.             int ms = me.getModifiersEx();
37.             String s = me.getModifiersExText(ms);
38.             System.out.println(s);
39.         }
40.         public void mouseReleased(MouseEvent me)
41.         {
42.             System.out.println("Mouse Released...");
43.             System.out.println("ClickCount—>" + me.getClickCount());
44.         }
45.         public void mouseClicked(MouseEvent me)
46.         {
47.             System.out.println("Mouse Clicked...");
48.             System.out.println("Button—>" + me.getButton());
49.         }
50.     }
51.     class C implements MouseMotionListener
52.     {
53.         public void mouseMoved(MouseEvent me)
54.         {
55.             System.out.println("Mouse Moved...");
56.         }
57.         public void mouseDragged(MouseEvent me)
58.         {
59.             System.out.println("Mouse Dragged...");
60.         }
61.     }
62.     class D implements ActionListener
63.     {
64.         public void actionPerformed(ActionEvent ae)
```

```

65.         {
66.             System.out.println ("Button Pressed...");
67.         } }

```

The ActionEvent Class:

It is a semantic event, which indicates that a component – defined action occurred. This high level event is generated by a component (such as a Button) when the component- specific action occurs (Such as being pressed). The event is passed to every Action Listener object that registered to receive such events using the component’s add action Listener method.

The object that implements the Action Listener interface gets this Action Event when the event occurs. The listener is therefore spared the details of processing individual mouse movements and mouse clicks, and can instead process a “meaningful” (semantic) event like “button pressed”.

Note : To invoke an ActionEvent on a Button using the Keyboard, use the space bar.

Methods:

String **getAction (Command)**

Returns the command string associated with this action.

int **getModifiers ()**

Returns the modifier Keys held down during this action event.

long **getWhen ()**

Returns the time stamp of when this event occurred.

Example 18.8:

This example illustrates how we can identify the key modifiers effective when and Action Event occurs.

```

1. import java.awt.*;
2. import java.awt.event *;
3. public class KeyModifierDemo extends Frame implements ActionListener
4.     {
5.         Button b;
6.         KeyModifierDemo ( )
7.         {
8.             b = new Button ("Button");
9.             b.addActionListener (this);
10.            setLayout (new FlowLayout ( ) );
11.            add (b);

```

```

12.         setBounds (100,100,200,200);
13.         setVisible (true);
14.     }
15.     public static void main (String args[])
16.     {
17.         new KeyModifierDemo ( );
18.     }
19.     public void actionPerformed (ActionEvent ae )
20.     {
21.         System.out.println ("Button Pressed");
22.         int ms = ae.getModifiers ( );
23.         if (ms & InputEvent.CTRL_ DOWN_ MASK)!=0)
24.         {
25.             System.out.println ("CTRL key was pressed");
26.         }
27.         if (ms & InputEvent.SHIFT_ DOWN_ MASK)!=0)
28.         {
29.             System.out.println ("Shift Key was pressed");
30.         }
31.         if (ms & InputEvent.ALT_ DOWN_ MASK)!=0)
32.         {
33.             System.out.println ("ALT key was pressed");
34.         }
35.     }
36. }

```

Handling Multiple Action Commands:

Action events are the simplest. When we find out that a scroll bar has sent an adjustment event, the obvious question is, "Now what is the scroll bar's value?". When we find out that the mouse has been clicked, it is natural to wonder, "What are the mouse's x and y coordinates?" But when we find out that a button has sent an Action event, there does not seem to be any additional information to seek; the button has been clicked, and that seems to be all there is to know.

However, there is an extra piece of information associated with an Action event. This information is a string, known as an Action command. We can extract an Action event's

Action command with the `getActionCommand ()` method. If an Action event was sent by a text field, `getActionCommand ()` returns the current contents of the text field. (A text field sends an Action event when the user types the Enter Key.). If an Action event was sent by a button, the default behavior is for `getActionCommand ()` to return the string that is the button's label. However, we can explicitly set a button's ActionCommand by calling its `setActionCommand (String)` method.

```
Button btn = new Button (“Hello”);
```

```
btn.setActionCommand (“Hello”);
```

The benefit of Action commands for buttons is apparent when a button's Action listener needs to decide how to react to an Action event. It is common for a single listener to act as a listener for several components ; in our context, a single object might be an Action listener for several or many button. In this situation, the object's `actionPerformed ()` method must be by determining which button was hit .The following example, demonstrates this:

Example 18.9:

```
1.      import java.awt.*;
2.      import java.awt.event.*;
3.      class ButtonRefTest
4.      {
5.          Button b1,b2,b3;
6.          ButtonRefTest()
7.          {
8.              Frame f=new Frame();
9.              f.setLayout(new FlowLayout ( ));
10.             f.setSize(300,300);
11.             b1=new Button(“Push Me”);
12.             b2=new Button(“Click Me”);
13.             b3=new Button(“Hit Me”);
14.             f.add(b1);          f.add(b2);          f.add(b3);
15.             B obj= new B(b1,b2,b3);
16.             b1.addActionListener(obj);
17.             b2.addActionListener(obj);
18.             b3.addActionListener(obj);
19.             f.setVisible(true);
20.         }
```

```

21.         public static void main(String args [ ])
22.         {
23.             new ButtonRefTest();
24.         }
25.     }
26.     class B implements ActionListener
27.     {
28.         Button b4,b5,b6;
29.         B(Button b1, Button b2, Button b3)
30.         {
31.             b4 = b1;
32.             b5 = b2;
33.             b6 = b3;
34.         }
35.         public void actionPerformed(ActionEvent ae)
36.         {
37.             Button b=(Button)ae.getSource();
38.             if(b==b4)
39.                 System.out.println("First Button was pressed");
40.             else if(b==b5)
41.                 System.out.println("Second Button was pressed");
42.             else if(b==b6)
43.                 System.out.println("Third Button was pressed");
44.         }
45.     }

```

Example 18.10:

```

1.     import java.awt.*;
2.     import java.awt.event.*;
3.     class GetLabelTest
4.     {
5.         Button b1,b2,b3;

```

```

6.         GetLabelTest()
7.         {
8.             Frame f=new Frame();
9.             f.setLayout(new FlowLayout ( ));
10.            f.setSize(300,300);
11.            b1=new Button("Push Me");
12.            b2=new Button("Click Me");
13.            b3=new Button("Hit Me");
14.            f.add(b1);          f.add(b2);          f.add(b3);
15.            B obj= new B();
16.            b1.addActionListener(obj);
17.            b2.addActionListener(obj);
18.            b3.addActionListener(obj);
19.            f.setVisible(true);
20.        }
21.        public static void main(String args [ ])
22.        {
23.            new GetLabelTest();
24.        }
25.    }
26.    class B implements ActionListener
27.    {
28.        public void actionPerformed(ActionEvent ae)
29.        {
30.            Button b=(Button)ae.getSource();
31.            String s=b.getLabel();
32.            if(s.equals("Push Me"))
33.                System.out.println("First Push Me Button was pressed");
34.            else if(s.equals("Click Me"))
35.                System.out.println("Second Push Me Button was pressed");
36.            else if(s.equals("Hit Me"))
37.                System.out.println("Third Push Me Button was pressed");

```

38. }

39. }

Example 18.11: Using `setActionCommand()` method to set the Action Command.

1. import java.awt.*;

2. import java.awt.event.*;

3. class ActionCommandTest

4. {

5. Button b1,b2,b3;

6. ActionCommandTest()

7. {

8. Frame f=new Frame();

9. f.setLayout(new FlowLayout());

10. f.setSize(300,300);

11. b1=new Button("Push Me");

12. b2=new Button("Push Me");

13. b3=new Button("Push Me");

14. b1.setActionCommand("First");

15. b2.setActionCommand("Second");

16. b3.setActionCommand("Third");

17. f.add(b1); f.add(b2); f.add(b3);

18. B obj= new B();

19. b1.addActionListener(obj);

20. b2.addActionListener(obj);

21. b3.addActionListener(obj);

22. f.setVisible(true);

23. }

24. public static void main(String args [])

25. {

26. new ActionCommandTest();

27. }

28. }

29. class B implements ActionListener

```

30.     {
31.         public void actionPerformed(ActionEvent ae)
32.         {
33.             Button b=(Button)ae.getSource();
34.             String s=b.getActionCommand();
35.             if(s.equals("First"))
36.                 System.out.println("First Push Me Button was pressed");
37.             else if(s.equals("Second"))
38.                 System.out.println("Second Push Me Button was pressed");
39.             else if(s.equals("Third"))
40.                 System.out.println("Third Push Me Button was pressed");
41.         }
42.     }

```

Handling Key Events

Example 18.12:

```

1.     import java.awt.*;
2.     import java.awt.event.*;
3.     class KeyTest extends Frame implements KeyListener
4.     {
5.         String msg = "";
6.         int x=50,y=100;
7.         public static void main (String arg[])
8.         {
9.             Frame f= new KeyTest();
10.            f.setSize(300, 300);
11.            f.addKeyListener((KeyTest)f);
12.            f.setVisible (true);
13.        }
14.        public void keyReleased(KeyEvent ke)
15.        {
16.            System.out.println("Key Up...");
17.        }

```

```
18.     public void keyTyped(KeyEvent ke)
19.     {
20.         System.out.println("Key Typed...");
21.         msg = msg + ke.getKeyChar();
22.         repaint();
23.     }
24.     public void keyPressed(KeyEvent ke)
25.     {
26.         System.out.println("Key Pressed...");
27.         int key = ke.getKeyCode();
28.         switch(key)
29.         {
30.             case KeyEvent.VK_F1:
31.                 msg = msg + "<F1>"; break;
32.             case KeyEvent.VK_F2:
33.                 msg = msg + "<F2>"; break;
34.             case KeyEvent.VK_F3:
35.                 msg = msg + "<F3>"; break;
36.             case KeyEvent.VK_PAGE_DOWN:
37.                 msg = msg + "<PgDn>"; break;
38.             case KeyEvent.VK_PAGE_UP :
39.                 msg = msg + "<PgUp>"; break;
40.             case KeyEvent.VK_LEFT:
41.                 msg = msg + "<Left Arrow>"; break;
42.             case KeyEvent.VK_RIGHT:
43.                 msg = msg + "<Right Arrow>"; break;
44.             case KeyEvent.VK_CONTROL:
45.                 msg = msg + "<Ctrl>"; break;
46.         }
47.         repaint();
48.     }
49.     public void paint(Graphics g)
```

```

50.         {
51.             g.setColor(Color.red);
52.             g.drawString(msg,x,y);
53.         }
54.     }

```

Example 18.13: Demonstration of FocusListener Color of TextField changes on focus lost or gained

```

1.     import java.awt.*;
2.     import java.awt.event.*;
3.     class FocusTest extends Frame implements FocusListener
4.     {
5.         static TextField tf1,tf2;
6.         public static void main(String args[])
7.         {
8.             FocusTest f = new FocusTest();
9.             f.setLayout(new FlowLayout());
10.            tf1 = new TextField(8);
11.            tf1.setBackground(Color.blue);
12.            tf1.addFocusListener(f);
13.            tf2 = new TextField(8);
14.            tf2.setBackground(Color.blue);
15.            tf2.addFocusListener(f);
16.            f.add(tf1);
17.            f.add(tf2);
18.            f.setSize(200,200);
19.            f.setVisible(true);
20.        }
21.        public void focusGained(FocusEvent fe)
22.        {
23.            TextField tf=(TextField)fe.getSource();
24.            tf.setBackground(Color.red);
25.        }
26.        public void focusLost(FocusEvent fe)

```

```

27.         {
28.             TextField tf=(TextField)fe.getSource();
29.             tf.setBackground(Color.blue);
30.         }
31.     }

```

Example 18.14: Demonstration of requestFocus() We can't use WindowListener in same class so we defined two classes one for frame & another for WindowListener.

```

1.     import java.awt.*;
2.     import java.awt.event.*;
3.     class A extends Frame implements ActionListener
4.     {
5.         static TextField tf1,tf2;
6.         static Button b1;
7.         public static void main(String args[])
8.         {
9.             A f = new A();
10.            f.setLayout(new FlowLayout());
11.            f.addWindowListener(new B());
12.            tf1 = new TextField(8);
13.            tf2 = new TextField(8);
14.            b1 = new Button("Set");
15.            b1.addActionListener(f);
16.            f.add(tf1);
17.            f.add(tf2);
18.            f.add(b1);
19.            f.setSize(200,200);
20.            f.setVisible(true);
21.        }
22.        public void actionPerformed(ActionEvent ae)
23.        {
24.            tf2.requestFocus();
25.        }
26.    }

```



```

27.     class B extends WindowAdapter
28.     {
29.         public void windowClosing(WindowEvent we)
30.         {
31.             Frame f=(Frame)we.getWindow();
32.             f.dispose();
33.         }
34.     }

```

Example 18.15 Displaying a dialog box

```

1.     import java.awt.*;
2.     class DialogTest
3.     {
4.         public static void main(String args[]) throws InterruptedException
5.         {
6.             Frame f1 = new Frame();
7.             f1.setSize(300,300);
8.             f1.setVisible(true);
9.             Dialog d1 = new Dialog(f1);
10.            d1.setSize(100,100);
11.            d1.setVisible(true);
12.            Thread.sleep(3000);
13.            d1.dispose();
14.        }
15.    }

```

Example 18.16 Demonstration of Menu Bar

```

1.     import java.awt.*;
2.     import java.awt.event.*;
3.     class MyMenu extends Frame implements ActionListener, WindowListener,
        ItemListener
4.     {
5.         TextField tf1;
6.         String msg="";

```

```
7.             CheckboxMenuItem item4;
8. MyMenu()
9.     {
10.         super("Matrix");
11.         setLayout(new FlowLayout());
12.         tf1 = new TextField(8);
13.         add(tf1);
14.         MenuBar mbar = new MenuBar();
15.         Menu file = new Menu("File");
16.         MenuItem item1 = new MenuItem("New...");
17.         item1.addActionListener(this);
18.         file.add(item1);
19.         MenuItem item2 = new MenuItem("Open...");
20.         item2.addActionListener(this);
21.         file.add(item2);
22.         Menu sub = new Menu("Sub Menu");
23.         MenuItem item3 = new MenuItem("Option1");
24.         sub.add(item3);
25.         file.add(sub);
26.         item4 = new CheckboxMenuItem("TextBox",true);
27.         item4.addItemListener(this);
28.         file.add(item4);
29.         mbar.add(file);
30.         addWindowListener(this);
31.         setMenuBar(mbar);
32.         setSize(400,400);
33.         setVisible(true);
34.     }
35.     public void paint(Graphics g)
36.     {
37.         g.drawString(msg,150,250);
38.     }
```

```
39.         public void actionPerformed(ActionEvent ae)
40.         {
41.             String s1=(String)ae.getActionCommand();
42.             if(s1.equals("New..."))
43.             {
44.                 tf1.setText("New Selected");
45.                 msg="New Selected";
46.             }
47.             else if(s1.equals("Open..."))
48.             {
49.                 tf1.setText("Open Selected");
50.                 msg="Open Selected";
51.             }
52.             repaint();
53.         }
54.     public void windowClosing(WindowEvent we)
55.     {
56.         Frame f = (Frame)we.getWindow();
57.         f.dispose();
58.     }
59.     public void windowClosed(WindowEvent we){}
60.     public void windowActivated(WindowEvent we){}
61.     public void windowDeactivated(WindowEvent we){}
62.     public void windowIconified(WindowEvent we){}
63.     public void windowDeiconified(WindowEvent we){}
64.     public void windowOpened(WindowEvent we){}
65.     public void itemStateChanged(ItemEvent ie)
66.     {
67.         tf1.setVisible(item4.getState());
68.     }
69.     public static void main(String args[])
70.     {
```

```
71.             new MyMenu();
72.         }
73.     }
```

Example 18.17 Demonstration of Listbox

```
1.     import java.awt.*;
2.     import java.awt.event.*;
3.     class ListTest extends Frame implements ActionListener
4.     {
5.         static List l1,l2;
6.         public static void main(String args[])
7.         {
8.             ListTest f = new ListTest();
9.             f.setLayout(new FlowLayout());
10.            l1 = new List(3,true);
11.            l1.add("India");
12.            l1.add("Pakistan");
13.            l1.add("USA");
14.            l1.add("UK");
15.            l1.add("China");
16.            l2 = new List(3,true);
17.            Button b1 = new Button("Transfer");
18.            b1.addActionListener(f);
19.            f.add(l1);
20.            f.add(l2);
21.            f.add(b1);
22.            f.setSize(200,200);
23.            f.setVisible(true);
24.        }
25.        public void actionPerformed(ActionEvent ae)
26.        {
27.            /* for(int i=l1.getItemCount()-1; i>=0; i—)
28.                {
```

```

29.                                     l2.add(l1.getItem(i));
30.                                     l1.remove(i);
31.                                     }
32.                                     */
33.                                     String s1[ ] = l1.getSelectedItems();
34.                                     for(int i=0;i<s1.length;i++)
35.                                     {
36.                                         l2.add(s1[i]);
37.                                         l1.remove(s1[i]);
38.                                     }
39.                                     }
40.                                     }

```

Example 18.18 Sum of two numbers using frame

```

1.     import java.awt.*;
2.     import java.awt.event.*;
3.     class Sum implements ActionListener
4.     {
5.         static Button b1,b2;
6.         static TextField tf1,tf2,tf3;
7.         static Frame f;
8.         public static void main(String args[])
9.         {
10.            f = new Frame();
11.            Sum s1 = new Sum();
12.            f.setLayout(new GridLayout(4,2));
13.            Label l1 = new Label("Enter No. 1:");
14.            tf1 = new TextField(8);
15.            Label l2 = new Label("Enter No. 2:");
16.            tf2 = new TextField(8);
17.            Label l3 = new Label("Result   :");
18.            tf3 = new TextField(8);
19.            b1 = new Button("Sum");

```

```
20.         b2 = new Button("Close");
21.         b1.addActionListener(s1);
22.         b2.addActionListener(s1);
23.         f.add(l1);
24.         f.add(tf1);
25.         f.add(l2);
26.         f.add(tf2);
27.         f.add(l3);
28.         f.add(tf3);
29.         f.add(b1);
30.         f.add(b2);
31.         f.setSize(200,200);
32.         f.setVisible(true);
33.     }
34.     public void actionPerformed(ActionEvent ae)
35.     {
36.         Button b=(Button)ae.getSource();
37.         if(b==b1)
38.         {
39.             int n1=Integer.parseInt(tf1.getText());
40.             int n2=Integer.parseInt(tf2.getText());
41.             tf3.setText(String.valueOf(n1+n2));
42.         }
43.         if(b == b2)
44.         {
45.             f.dispose();
46.         }
47.     }
48. }
```

Example 18.19 Displaying an image on Applet

```
1.      import java.awt.*;
2.      import java.net.*;
3.      import java.applet.*;
4.      /*<Applet code="ScrollPane1" width=200 height=200></applet>*/
5.      public class ScrollPane1 extends Applet
6.      {
7.          Image img1;
8.          URL file1;
9.          public void init()
10.         {
11.             try
12.             {
13.                 file1 = new URL("FILE:///c:/javaprg/a.jpg");
14.             }
15.             catch(MalformedURLException e)
16.             {
17.                 System.out.print(e);
18.             }
19.             img1=getImage(file1);
20.         }
21.         public void paint(Graphics g)
22.         {
23.             g.drawImage(img1,0,0,this);
24.         }
25.     }
```

Example 18.20 Demonstration of moving circle

```
1.      import java.awt.*;
```

```
2.     import java.awt.event.*;
3.     class Circle1 extends Frame implements MouseListener,Runnable
4.     {
5.         int x,y,w,h;
6.         static Circle1 f;
7.         public void run()
8.         {
9.             y=50;
10.            w=50;
11.            h=50;
12.            for(x=0;x<500 ;x+=10)
13.            {
14.                try
15.                {
16.                    Thread.sleep(500);
17.                }
18.                catch(InterruptedException e)
19.                {
20.                    System.out.println(e);
21.                }
22.                repaint();
23.            }
24.        }
25.        public static void main(String args[])
26.        {
27.            f = new Circle1();
28.            Thread t=new Thread(f);
29.            t.start();
30.            f.addMouseListener(f);
31.            f.setSize(500,500);
32.            f.setVisible(true);
33.        }
```



```
34.         public void paint(Graphics g)
35.     {
36.         g.setColor(Color.RED);
37.         g.drawOval(x,y,w,h);
38.         g.setColor(Color.BLUE);
39.         g.fillOval(x+1,y+1,w-1,h-1);
40.     }
41.     public void mousePressed(MouseEvent me)
42.     {
43.         x = me.getX();
44.         y = me.getY();
45.         w=50;
46.         h=50;
47.         Graphics g=f.getGraphics();
48.         g.clearRect(0,0,200,200);
49.         paint(g);
50.     }
51.     public void mouseReleased(MouseEvent me){}
52.     public void mouseClicked(MouseEvent me){}
53.     public void mouseEntered(MouseEvent me){}
54.     public void mouseExited(MouseEvent me){}
55. }
```





CHAPTER

∞ 19 ∞

(Painting in AWT)

Introduction-

Many types of AWT components (buttons and scrollbars, for example) have their appearance dictated by the underlying window system. Other component types notably applets, frames, panels and canvases have no intrinsic appearance. If we use any of these classes other than simply as containers and want our component to look at all useful, we will have to provide the code that implements the components appearance.

Every component has a graphics context associated with it. The graphics context correspondence to the frame buffer related to the component.

We can draw on the component using methods supported by the Graphics Context. Anything drawn on the Graphics context appears on the component.

The correct approach for painting on any component is to override its paint() method and provide the appropriate code.

The paint() method has one argument of type Graphics, which represents the Graphics Context of the component. It is also possible to draw on the component without using paint () method although this approach is not recommended.

The paint () method and the Graphics Context :

One interesting point about the frame in the following example is that no explicit

calls made to the `paint()` method; the method is simply provided. The environment seems to do a good job of calling `paint()` at the right moment.

Painting on a component is accomplished by making calls to a Graphics context, which is an instance of the Graphics class. A graphics context knows how to render onto a single target. The three media a Graphics Context can render onto are:





1. **Components.**
2. **Images.**
3. **Printers.**

Any Kind of component can be associated with a Graphics context. The association is permanent; a context cannot be reassigned to a new component. Although we can use graphics contexts to paint onto any kind of component, it is unusual to do so with components that already have an appearance.

Buttons, Choices, Checkboxes, Labels, Scrollbars, TextFields, and TextAreas do not often require programmer level rendering. Most often, these components just use the version of `paint()` that they inherit from the component super class.

This version does nothing; the components are rendered by the underlying window system. However, there are four classes of “blank” components that have no default appearance and will show up as empty rectangles, unless they are sub-classed and given `paint()` methods.

These four component classes are:

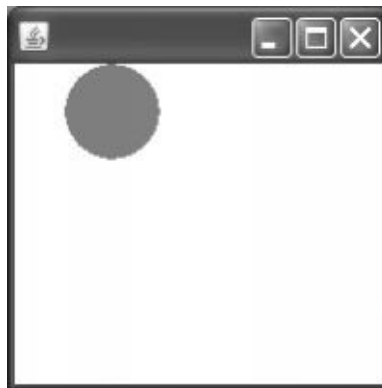
-  Applet.
-  Canvas.
-  Frame.
-  Panel.

Example19.1 The Frame Class is extended and `paint()` methods is over-ridden to change the appearance of the frame.

```
1.      import java.awt.*;
2.      public class SimplePaint extends Frame
3.      {
4.          public static void main (String args[])
5.          {
6.              Frame f=new SimplePaint();
7.              f.setSize(200,200);
```

```
8.             f.setVisible(true);
9.         }
10.        public void paint (Graphics g)
11.        {
12.            g.setColor(Color.white);
13.            g.fillRect(0,0,300,300);
14.            g.setColor(Color.grey);
15.            g.fillOval(30,30,50,50);
16.        }
17.    }
```

Output:



When we subclass a component class and give the sub class its own paint() method, the environment calls that method at appropriate times, passing in an appropriate instance of Graphics.

Operations supported by the graphics class

The four major categories of operations provided by the graphics class (graphics context) are:

- ✚ Selecting a color.
- ✚ Selecting a font.
- ✚ Drawing and filling.
- ✚ Clipping.

Selecting a color:

Colors are selected by calling the setColor() method. The required argument is an instance of the color class.

There are 13 pre-defined colors, accessible as static final variables of the color class.




(The variables are themselves instance of the color class. Which makes some people uneasy, but java has no trouble with such things.) The pre-defined colors are:

1. **Color.black**
2. **Color.blue**
3. **Color.cyan**
4. **Color.darkGray**
5. **Color.gray**
6. **Color.green**
7. **Color.lightGray**
8. **Color.magenta**
9. **Color.orange**
10. **Color.pink**
11. **Color.red**
12. **Color.white**
13. **Color.yellow**

If we want a color that is not on this list, we can construct our own. There are several version of the color constructor,the simplest is:

Color(int redLevel, int greenLevel, int blueLevel)

The three parameters are intensity level on a scale of 0 to 255 , for the primary colors. The colors are additive, which means they mix like light, not like paint.

- | | |
|--|-------------------------------------|
|  new Color(0,0,0) | black(absence of light) |
|  new Color(255,0,0) | red (only red hue is included) |
|  new Color(255,255,255) | white (all colors shining at once) |

Code fragment below lists the first part of a paint() method that sets the color of its graphics context to pale green:

```
public void paint(Graphics g)
{
    Color c= new Color(170,255,170);
    g.setColor(c) ;
}
```

After call to setColor() in the above code all graphics will be painted in pale (light)green, until the next g.setColor() call. Calling g.setColor(0 does not change the color of anything that has already been drawn; it only affect subsequent operations.

Selecting a Font:

Setting the font of a graphics context is like setting the color subsequent string drawing operations will use the new font, while previously drawn strings are not affected. Before we can set a font, we have to create one. The constructor for the font class looks like this:

Font(string fontName, int style, int size)

The first parameter is the name of the font. Font availability is platform dependent. We can get a list of available font names, returned as an array of strings , by calling the getFontList()method on our toolkit like this:

String fontNames[]= Toolkit.getDefaultToolkit().getFontList();

There are three font names that are platform independent and that we are encouraged to use:

1. “Serif”
2. “Sansserif”
3. “Monospaced”

On 1.0.X releases of the JDK these were called, respectively “TimesRoman”,”Helvetica”and “Courier”.

The style parameter can be sent to any of the following int constant:

1. **Font.PLAIN**
2. **Font.BOLD**
3. **Font.ITALIC**

The code fragment below sets the fonts of graphics context gc to 24 point bold sansserif:

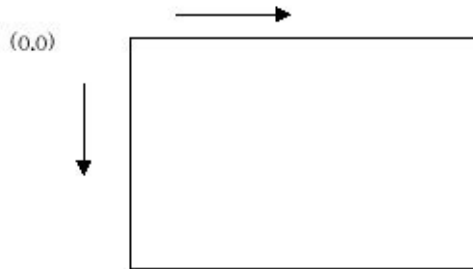
```
Font f =new Font(“SansSerif”,Font.BOLD,24);  
gc.setFont(f);
```

We can specify combinations of styles, for example a bold italic font, by passing the sum if styles, like this:

Font.BOLD + Font.ITALIC

Drawing and filling:

All the rendering methods of graphics class specify pixel co-ordinate positions for the shapes they render every component has its won co-ordinate space, with the origin in the component’s upper-left corner, x increasing to the right and y increasing downward. Following figure shows the component co-ordinate system:



The graphics context class does not have an extensive set of painting methods. (Sophisticated rendering is handled by extended APIs such as 2d,3d,and animation.)

The methods in the graphic context class that we need to know about are:

void drawLine(int x1,int y1, int x2, int y2)

Draws a line, using the current color, between the points (x1,y1) and (x2, y2) in this graphics context's coordinate system.

void drawRect(int x, int y, int width, int height)

Draws the outline of the specified rectangle.

void fillRect (int x, int y, int width, int height)

Fills the specified rectangle.

void drawOval(int x, int y, int width, int height)

Draws the outline of an oval bounded by he specified rectangle.

void fillOval(intx,int y, int width, int height)

Fills an oval bounded by the specified rectangle with the current color.

An oval is specified by a rectangular bounding box. The oval lies inside the bounding box and is tangent to each of the box's sides at the mid-point as shown below:

To draw a circle, use a square-bounding box. Note that the painting only draws foreground pixels, not background, so the space inside the bounding box but outside the oval is left unchanged. Here x and y are the co-ordinates of the upper left corner of the bounding box, and width and height are the width and height of the box.

void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)

Draw the outline of a circular or elliptical arc covering the specified rectangle.

void fillArc(int x, int y, int width, int height, int startangle, int arcangle)

Fills a circular or elliptical arc covering the specified rectangle.

An arc is segment of an oval. To specify an arc, we first specify the oval's bounding box, just as we do with drawOval() and fillOval(). We also need to specify the starting and ending points of the arc, which we do by specifying a starting angle and the angle swept out by the arc.

A filled arc is region bounded by the arc itself and the two radii from the center of the oval to the end points of the arc.

void drawPolygon(int[] xPoints, int[] yPoints, int nPoints)

Draws a closed polygon defined by arrays of x and y coordinates.

void fillPolygon (int[] xPoints , int[] yPoints, int nPoints)

Fills a closed polygon defined by arrays of x and y coordinates.

A polygon is a closed figure with an arbitrary number of vertices. The vertices are passed to the drawPolygon() and fillPolygon() methods as two int arrays. The first array contains the x co-ordinates of the vertices, the second array contains the y co-ordinates. A third parameter specifies the number of vertices.

Note: end point is joined with the first point.

void drawPolyLine(int[] xPoints, int[] yPoints , int nPoints)

Draws a sequence of connected lines defined by array of x and y coordinates.

A polyline is similar to a polygon , but it is open rather than closed. There is no line segment connecting the last vertex to the first.

void drawString(String str, int x, int y)

Draws the text given by the specified string, using this graphics context's current font and color. The x , y parameters specify the left edge of the baseline of the string. The

characters with descenders(g, j, p,q, and y in most fonts)extend below the baseline.

The fact that text-coordinates are relative to the baseline is important. In a simple text, for example, we might try to draw a string at (0,0) expecting it to appear at the top left of the space. However, because of the baseline, we will only see the descenders of the string, which might mean that we see nothing at all.

By default, a graphics context uses the font of the associated component however, we can set a different font by calling the graphics context's setFont () method.

boolean drawImage (Image img, int x, int y, ImageObserver observer)

Draws as much of the specified image as is currently available.

Here, im is the image to be rendered, and x and y are the coordinates within the destination component of the upper-left corner of the image. The image observer must be an object that implements the imageobserver interface.

An image is an off-screen representation of a rectangular collection of pixel values. Java's image support is complicated.

For now, assume that we have somehow obtained an image (that is, an instance of class java.awt,image) that we want to render to the screen using a certain graphics context. The way to do this is to call the graphics context's drawImage()method.

Example19.2:

This example illustrates how to create an image and then draw on it using its graphics context . the image can then be displayed in the applet window using drawImage()method.

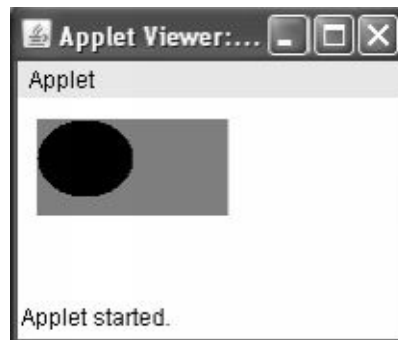
```
1.      import java.awt.*;
2.      import java.applet.*;
3.      import java.net.URL;
4.      /*<applet code="PaintImage" width=200 height=100></applet>*/
5.      public class PaintImage extends Applet
6.      {
7.          private Image im;
8.          public void init()
9.          {
```

```

10.         im = createImage(100,50);
11.         Graphics imgc=im.getGraphics();
12.         imgc.setColor(Color.gray);
13.         imgc.fillRect(0,0,100,50);
14.         imgc.setColor(Color.black);
15.         imgc.fillOval(0,0,50,40);
16.     }
17.     public void paint(Graphics g)
18.     {
19.         g.drawImage(im,10,10,this);
20.     }
21. }

```

Output:



Example19.3:

This example illustrates how to create an image object from an existing image. The image can then be displayed in the applet window using drawimage() method.

```

1.     import java.awt.*;
2.     import java.applet.*;
3.     import java.net.*;
4.     /*
5.     <applet code="PaintImage" width=500 height=500>
6.     </applet>
7.     */
8.     public class PaintImage 1 extends Applet
9.     {
10.         private Image im;

```

```

11.         public void init()
12.             {
13.     try
14.             {
15.                 im=getImage(new URL("FILE:///c:/matrix/logo.gif"));
16.                 //im=getImage(newURL("http://127.0.0.1:8008/logo.gif"));
17.             }
18.         catch (MalformedURLException e)
19.             {
20.                 System.out.println(e);
21.             }
22.         }
23.     public void paint(Graphics g)
24.         {
25.             g.drawImage(im,10,10,this);
26.         }
27.     }

```

Clipping:

Most calls that programmers make on graphics context involve color selection or drawing and filling. A less common operation is clipping. Clipping is simply restricting the region that a graphics context can modify.

Every graphics context that is, every instance of the graphics class has a clip region, which defines all or part of the associated component. When we call one of the drawXXX() or fillXXX() method of the graphics class, only those pixels that lie within the graphics context's clip region are modified.

The default clip region for a graphics context is the entire visible region of the associated component. There are methods that retrieve and modify a clip region. Consider the following paint method.

Example 19.4:

```

1.     import java.awt.*;
2.     class Clip extends Frame
3.     {
4.         public static void main(String args[])
5.         {

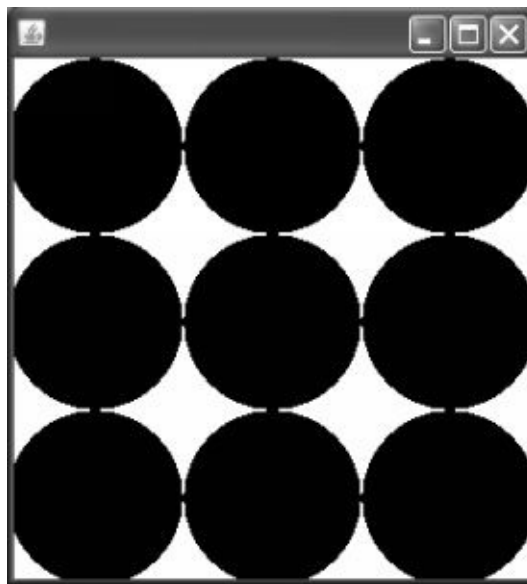
```

```

6.         Frame f=new Clip();
7.         f.setSize(300,330);
8.         f.setVisible(true);
9.     }
10.    public void paint(Graphics g)
11.    {
12.        for(int i=0; i<300; i=i+100)
13.        {
14.            for(int j=30; j<330; j=j+100)
15.            {
16.                g.fillOval(i,j,100,100);
17.            }
18.        }
19.    }
20. }

```

Output:



The paint () method of above program draws a dot pattern. Consider what happens when this is the paint() method of an applet/ frame that is 200 * 200 pixels. Because loop counters go all the way up to 500, the method attempts to draw outside the bounds of the applet/ frame.

This is not a problem, because the graphics context by default has a clip region that coincides with the applet / frame itself.

To set a rectangular clip region for a graphics context, we can call the set Clip (x,y, width, height) method, passing in four ints that describe the position and size of the

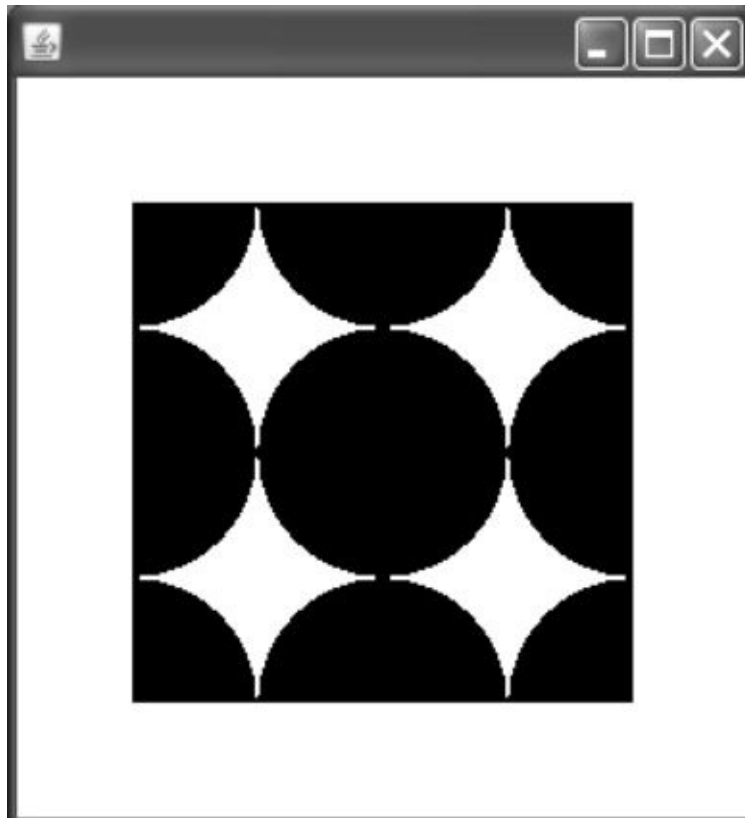
desired clip rectangle.

For example, the code above could be modified as follows:

Example 19.5:

```
1.     import java.awt.*;
2.     class Clip2 extends Frame
3.     {
4.         public static void main(String args[])
5.         {
6.             Frame f=new Clip2();
7.             f.setSize(300,330);
8.             f.setVisible(true);
9.         }
10.        public void paint(Graphics g)
11.        {
12.            g.setClip(50,80,200,200);
13.            for(int i=0; i<300; i=i+100)
14.                {
15.                    for(int j=30; j<330; j=j+100)
16.                        {
17.                            g.fillOval(i,j,100,100);
18.                        }
19.                }
20.            g.drawString ("Hello! Kamal", 50,50);
21.            g.drawString ("Bye! Kamal", 120,150);
22.        } }
```

Output:



Now painting is clipped to a 200 * 200 rectangle in the center of the 300 * 300 applet frame as shown above.

Clipping is good to Know about in its own right. Clipping also comes into play when the environment needs to repair exposed portions of a component.

Summary about the paint () method and the graphics context:

A Graphics context is dedicated to single component. To Paint on a component, we call the graphics context's drawXXXX () and fillXXXX () methods. To change the color or font of graphics operations, we call the graphics context's setColor () or setFont () methods, respectively.

GUI Thread and painting:

The Java runtime environment creates and controls its own threads that operate behind the screens, and one of the threads is responsible for GUI management.

The GUI thread is the environment's tool for accepting user input events and for calling the paint () method of components that need painting.

Call to paint () are not all generated by the environment. Java programs can of course make their own calls, either directly or indirectly via the repaint () method.

It is also possible to draw on a component using its graphics context. There is no need to call paint () or repaint () method in this approach.

Different approaches for drawing on a component are as follows:

- (i) Directly drawing after obtaining Graphics Context the component.
- (ii) Spontaneous painting (paint () method is called implicitly/ automatically by GUI thread)
- (iii) Explicitly invoking paint () method
- (iv) Explicitly invoking repaint () method, which then indirectly invokes the paint () method ().

Directly drawing after obtaining Graphics Context of the component:

Since it is possible to get a reference to a component's Graphics context using the method `getGraphics ()`, it is tempting to believe that one can simply draw to a component from any convenient method. Generally this approach is seriously flawed.

Suppose we have an applet that wants to draw a red dot at the point of the most recent mouse click. T

he remainder of the applet should be yellow. Assume that the applet is handling its own mouse events. We might consider creating an event handler like this:

Example 19.6:

```

1.      import java.awt.*;
2.      import java.awt.event.*;
3.      class PaintApproach extends Frame implements MouseListener
4.      {
5.          public static void main (String args [])
6.          {
7.              PaintApproach f = new PaintApproach ( );
8.              f.setSize (300,300);
9.              f.addMouseListener(f);
10.             f.setVisible(true);
11.         }
12.         public void mouseEntered (MouseEvent e)      { }
13.         public void mouseExited (MouseEvent e)      { }
14.         public void mousePressed (MouseEvent e)     { }
15.         public void mouseReleased (MouseEvent e)    { }
16.         public void mouseClicked (MouseEvent e)
17.         {

```

```

18.         System.out.println ("Mouse Clicked...");
19.         Graphics g = getGraphics ( );
20.         g.setColor(Color.yellow);
21.         g.fillRect(0,0, getSize().width, getSize().height);
22.         g.setColor (Color.Red);
23.         g.fillOval (e.getX() -10, e.getY()-10,20,20);
24.     }
25. }

```

If this applet ever gets covered and exposed, the GUI thread will implicitly call `paint ()`, unfortunately, `paint ()` does not know about the red circle that was drawn in method `mouse Clicked ()`, so the red circle will not be repaired.

The proper place for all drawing operations is in `paint ()`, or in methods called from `paint ()`; this technique ensures that the GUI thread will be able to repair exposure damage. The GUI thread expects `paint ()` to be able to correctly reconstruct the screen at any arbitrary moment.

One possible way to achieve this is to remove all drawing code from event handlers and other arbitrary methods and place that code into the `paint ()` method.

Next, when a method needs to draw, arrange for it to write to variables that describe the desired drawing, so that those variables are accessible to the `paint ()` method and can be used by the `paint ()` method to perform the drawing.

Now when the `paint ()` method gets called, whether by one of our general methods, or by the GUI thread, it can draw whatever was originally intended.

Spontaneous Painting (`paint()` method is called implicitly/automatically by GUI thread):

Spontaneous painting is not an official Java term, but it gets the point across. Some painting happens by itself, with no impetus by the program. For example, as every introductory

Java book explains, when a browser starts-up an applet, shortly after the `init ()` method completes, call is made to the `paint ()` method. Also, when part or all of a browser or a frame is covered by another window and then becomes exposed a call is made to the `paint ()` method so as to refresh the applet window.

It is the GUI thread that executes these calls to `paint ()`. Every applet and every application that has a GUI, has a GUI thread.

The GUI thread spontaneously calls `paint ()` whenever all or part of a component needs redrawing after any of the following events:

- After exposure.

✚ After de- iconification.

✚ After first display.

When a browser returns to a previously page containing an applet, provided that applet is at least partially visible. When the GUI thread calls `paint ()`, it must supply a graphics context, since the `paint ()` method's input parameter is an instance of the `Graphics` class.

An earlier section (“clipping”) discussed the fact that every graphics context has a clip region. The GUI thread makes sure that the graphics contexts that get passed to `paint ()` have their clip regions appropriately set. Most often, the default clip region is appropriate.

However, when a component is exposed, the clip region is set to be just that portion of the component that requires repair. If only a small piece of the component was exposed, then the clip region ensures that no time is wasted drawing pixels that are already the correct color.

If an exposed region is non- rectangular, then multiple calls to `paint ()` might occur, with the clip region of each successively building up to ensure that the whole exposed area gets repainted eventually.

Explicitly invoking `paint ()` method:

The component may not get refreshed if the drawing operations on it are performed in a method other than the `paint ()` method. If drawing is done in the `paint ()` method then it gets displayed when we call the `paint ()` method explicitly as well as when the `paint ()` method is called implicitly GUI thread.

The mouse Clicked () method of the previous example is rewritten so that it simply modifies the instance variables `mouse X` and `mouse Y` whenever mouse click happens.

The red dot is drawn in the `paint ()` method only so that it gets repaired even if applet gets covered and exposed.

Example 19.7:

The example in the previous section is modified so that the components display gets refreshed in the situations described above. This happens because the red dot is drawn in the `paint ()` method.

```
1.      import java.awt.*;
2.      import java.awt.event.*;
3.      public class PaintApproach1 extends Frame implements MouseListener
4.      {
5.          int mouseX, mouseY;
6.          public static void main (String arg[])
```

```

7.      {
8.          PaintApproach1 f = new PaintApproach2 ( );
9.          f.setSize (300, 300);
10.         f.addMouseListener(f);
11.         f.setVisible(true);
12.     }
13.     public void mouseEntered (MouseEvent e)  {}
14.     public void mouseExited (MouseEvent e)  {}
15.     public void mousePressed (MouseEvent e)  {}
16.     public void mouseReleased (MouseEvent e)  {}
17.     public void mouseClicked (MouseEvent e)
18.     {
19.         System.out.println (“Mouse Clicked...”);
20.         mouseX = e.getX();
21.         mouseY = e.getY();
22.         Graphics g = getGraphics ( );
23.         paint(g);
24.     }
25.     public void paint (Graphics g)
26.     {
27.         g.setColor (Color.yellow);
28.         g.fillRect(0,0, getSize( ).width, getSize( ).height);
29.         g.setColor(Color.red);
30.         g.fillOval(mouseX-10,mouseY-10,20,20);
31.     }
32.     }

```

Explicitly invoking repaint() method, which then indirectly invokes the paint()method:

Invoking paint () method explicitly may also lead to two serious problems.

The first problem is that the GUI thread is calling the paint () method as and when it needs to. It occurs when exposure needs to be handled. This can cause thread interaction issues.

To be fair, we will see that we might have to handle some of these issues any way, even when using the preferred approach, since the `paint ()` method will use data that we are modifying in other threads. However, we will minimize the problem if we allow `paint ()` to be called only in the GUI thread.

The Second problem is one of CPU usage. Consider this scenario: our program is getting regular notification that it should update the display with the next frame of an animation or similar continuously moving graphic.

Also suppose that the drawing is complex and time- consuming. If we simply call the whole `paint ()` method, we might be only part way through the drawing when frame trigger occurs.

This would result in our program spending. Or trying to spend, all its time in drawing. Further, the drawing would get further and further behind where it should be.

These problems can be dealt simply by using the preferred method of handling program initiated drawing. The idea is that the main program maintains data set that describes the drawing that should be presented to the user, but does not directly call the drawing routines.

Instead, the program requests that the GUI thread should run the painting routines, in broadly the same way that thread would do if it were handling exposure.

The GUI thread uses the data set presented by main program to decide what and how to draw.

To support this approach, the GUI system provides a method called `repaint()`. This method is defined in the `Component` class, and so is available on anything visible in a GUI. When we call this method, we are issuing a request to the GUI thread that it should perform the painting system routines.

The GUI thread does so within a reasonable time scale, although this might be limited by other high- priority thread activity. This approach ensures that the thread interactions between the GUI thread and user threads are minimized.

We must, naturally, ensure that our data set is never used by the GUI thread while that data is in an inconsistent state. However, this can be achieved quite easily using either synchronization or double buffering technique.

Another consequence of this behavior is that, no matter how much drawing the program tries to do, the system generally remains synchronized with current frame.

This happens because if we call the `repaint ()` method ten times, for example, before the GUI thread is able to service the first call, then the result will be a single execution of the painting system.

Therefore, in overloaded conditions, the painting system automatically skips frames that the host CPU simply cannot deal with. This means that the programmer can write code that reliably gets the best available video performance on a given platform, but still works tolerably well on any other platform. The following code implements this kind of scheme.

Example 19.8:

```
1.      import java.awt.*;
2.      import java.awt.event .*;
3.  public class PaintApproach2 extends Frame implements MouseListener
4.      {
5.          int mouseX, mouseY;
6.          public static void main (String args [])
7.          {
8.              PaintApproach2 f = new PaintApproach2( );
9.              f.setSize (300,300);
10.             f.addMouseListener (f);
11.             f.setVisible(true);
12.         }
13.         public void mouseEntered (MouseEvent e)      { }
14.         public void mouseExited (MouseEvent e)      { }
15.         public void mouse Pressed (Mouse Event e)  { }
16.         public void mouse Released (Mouse Event e) { }
17.         public void mouseClicked(MouseEvent e)
18.         {
19.             System.out.println ("Mouse Clicked ... ");
20.             mouseX = e.getX ( );
21.             mouseY = e.get Y ( );
22.             repaint ( );
23.         }
24.         public void paint (Graphics g)
25.         {
26.             g.setColor (Color.white);
27.             g.fillRect (0,0,getSize ( ).width, getSize ( ).height );
28.             g.setColor (Color.gray );
29.             g.fillOval (MouseX-10 , mouseY -10,20,20);
30.         }
31.     }
```

Notice that `repaint()` call has replaced the direct handling of painting. Now, regardless of how frequently the event handler is called, calls to `paint()` will not outstrip the system's ability to draw, and the program can not fall increasingly behind.

The previous code shows the essence of the preferred approach to program initiated drawing. The main program stores information in instance variables and then calls the `repaint()` method.

The `repaint()` method requests that the GUI thread draws the screen. The GUI thread uses the `paint()` method to do the drawing, and that method should perform the drawing according to the information in the instance variable.

The benefits of this approach are:

The screen is correctly repainted when the environment spontaneously calls `paint()`.

The thread interactions, between foreground threads trying to draw and the GUI thread, are controlled and predictable.

The virtual machine is never overwhelmed by painting.

When all the code is written this way, there are no surprises for other programmers trying to debug the code.

Although this approach works well, there is a situation where still further improvement is needed: animation. If we use the approach exactly as described, we will probably notice that an animation flickers unpleasantly while it runs.

Smooth Animation:

One feature of the default behavior of the repainting mechanism that we might not always want is that the system clears our drawing area as part of the preparation to respond to our call to `repaint()`. This can make animations and other real time drawings flicker unpleasantly. It turns out that *when we call `repaint()`, the GUI thread does not call `paint()`; instead it calls the `update()` method. The default behavior of `update()` is to do two things. Clear the background and then call `paint`.* Like this:

```
public void update (Graphics g)
{
    g.clearRect (0,0,width, height);
    paint(g);
}
```

We might reasonably ask why `repaint()` starts by clearing the window if this causes flickering. Well, in many cases, drawings are done using lines, rectangles, ovals and so forth drawn on a background.

If we are trying to make a spinning stick, we need to remove the old line before we draw the new one. Another way to think about this is to consider that if we are trying to draw a new drawing, we should first erase the old one.

So, how can we arrange for the new drawing to appear without having to clear the whole display first? We simply need to find another way to remove the old drawing. The general approach is to ensure that we draw the whole drawing- both background and foreground each time we draw a frame. If we can arrange this, then we do not need to clear the component, and the flickering goes away.

To prevent the component being cleared before paint () is called, we simply override the update () method so that it calls paint directly, as shown below:

```
public void update (Graphics g)
{
    paint (g);
}
```

This is a standard technique and works perfectly, provided that our paint () method does indeed refresh all the pixels of the display. One particularly easy way to handle this is to use an off screen image to store a drawing of what should be shown on the display.

This technique is not always the most efficient, in either speed or memory terms, but it is simple and easy to implement, and generally well understood by Java programmers.





CHAPTER

∞ 20 ∞

(Java.Lang.Object Class)

Introduction-

The Object class of java.lang package is the root of all the hierarchies in Java. All classes extend the Object class, either directly or indirectly.

Even if you write a simple class (i.e. it is not extending any base-class), it implicitly extends the built-in Object class. Thus the features of this class will be available in all the classes.

Methods in Object class:

The object class defines following methods, which are inherited by all the classes.

1. **public int hashCode()**
2. **public boolean equals(Object obj)**
3. **public final Class getClass()**
4. **public String toString()**
5. **protected void finalize()** throws Throwable
6. **protected Object clone()** throws CloneNotSupportedException

7. public final void **wait**(long timeout) throws InterruptedException
8. public final void **wait**(long timeout, int nanos) throws InterruptedException
9. public final void **wait**() throws InterruptedException
10. public final void **notify**()
11. public final void **notifyAll**()

1. **public int hashCode():**

When storing objects in hash tables, this method can be used to get a hash value for an object. This value is guaranteed to be consistent during the execution of the program.

2. **boolean equals(Object obj):**

If every object is to be considered unique, then it is not necessary to override the equals() method of the Object class. This method compares object reference for equality.

3. **final Class getClass():**

Returns the runtime class of the object, which is represented by an object of the class java.lang.Class at runtime.

Example 20.1:The following example illustrates that this method can be used to get the Class object corresponding to any java object. We can then use methods of the class Class to get information about the object's class using reflection/introspection. getMethods() and getFields() will display all the public members only, but getDeclaredMethods() and getDeclaredFields() will display all the members.

```

1.     import java.lang.reflect.Method;
2.     import java.lang.reflect.Field;
3.     class DispClassMembers1
4.     {
5.         public static void main(String args[])
6.         {
7.             String s = new String("Hello");
8.             Class c = s.getClass();
9.             Method m[] = c.getMethods();
10.        System.out.println(".....Public Methods (" + m.length + ")......");
11.            for(int i=0; i < m.length; i++)
12.                System.out.println(m[i]);
13.            Field f[] = c.getFields();
14.            System.out.println(".....Public Fields (" + f.length + ")......");

```



```

15.                for(int i=0; i< f.length; i++)
16.                System.out.println(f[i]);
17.                m = c.getDeclaredMethods();
18.                System.out.println(".....Declared Methods (" + m.length + ").....");
19.                for(int i=0; i< m.length; i++)
20.                System.out.println(m[i]);
21.                f = c.getDeclaredFields();
22.                System.out.println(".....Declared Fields (" + f.length + ").....");
23.                for(int i=0; i< f.length; i++)
24.                System.out.println(f[i]);
25.                }
26.        }

```

Output:

.....Public Methods (72).....

```

public int java.lang.String.hashCode()
public int java.lang.String.compareTo(java.lang.String)
public int java.lang.String.compareTo(java.lang.Object)
public int java.lang.String.indexOf(int,int)
public int java.lang.String.indexOf(int)
public int java.lang.String.indexOf(java.lang.String)
public int java.lang.String.indexOf(java.lang.String,int)
public boolean java.lang.String.equals(java.lang.Object)
public java.lang.String java.lang.String.toString()
public char java.lang.String.charAt(int)
public int java.lang.String.codePointAt(int)
public int java.lang.String.codePointBefore(int)
public int java.lang.String.codePointCount(int,int)
public int java.lang.String.compareToIgnoreCase(java.lang.String)
public java.lang.String java.lang.String.concat(java.lang.String)
public boolean java.lang.String.contains(java.lang.CharSequence)
public boolean java.lang.String.contentEquals(java.lang.StringBuffer)
public boolean java.lang.String.contentEquals(java.lang.CharSequence)

```

```
public static java.lang.String java.lang.String.copyOf(char[])
public static java.lang.String java.lang.String.copyOf(char[],int,int)
public boolean java.lang.String.endsWith(java.lang.String)
public boolean java.lang.String.equalsIgnoreCase(java.lang.String)
public static java.lang.String java.lang.String.format(java.lang.String,java.lang.Object[])
public static java.lang.String
java.lang.String.format(java.util.Locale,java.lang.String,java.lang.Object[])
public byte[] java.lang.String.getBytes()
public byte[] java.lang.String.getBytes(java.nio.charset.Charset)
public byte[] java.lang.String.getBytes(java.lang.String) throws
java.io.UnsupportedEncodingException
public void java.lang.String.getBytes(int,int,byte[],int)
public void java.lang.String.getChars(int,int,char[],int)
public native java.lang.String java.lang.String.intern()
public boolean java.lang.String.isEmpty()
public int java.lang.String.lastIndexOf(int)
public int java.lang.String.lastIndexOf(java.lang.String)
public int java.lang.String.lastIndexOf(java.lang.String,int)
public int java.lang.String.lastIndexOf(int,int)
public int java.lang.String.length()
public boolean java.lang.String.matches(java.lang.String)
public int java.lang.String.offsetByCodePoints(int,int)
public boolean java.lang.String.regionMatches(boolean,int,java.lang.String,int,int)
public boolean java.lang.String.regionMatches(int,java.lang.String,int,int)
public java.lang.String
java.lang.String.replace(java.lang.CharSequence,java.lang.CharSequence)
public java.lang.String java.lang.String.replace(char,char)
public java.lang.String java.lang.String.replaceAll(java.lang.String,java.lang.String)
public java.lang.String java.lang.String.replaceFirst(java.lang.String,java.lang.String)
public java.lang.String[] java.lang.String.split(java.lang.String,int)
public java.lang.String[] java.lang.String.split(java.lang.String)
public boolean java.lang.String.startsWith(java.lang.String,int)
public boolean java.lang.String.startsWith(java.lang.String)
```

```
public java.lang.CharSequence java.lang.String.subSequence(int,int)
public java.lang.String java.lang.String.substring(int,int)
public java.lang.String java.lang.String.substring(int)
public char[] java.lang.String.toCharArray()
public java.lang.String java.lang.String.toLowerCase()
public java.lang.String java.lang.String.toLowerCase(java.util.Locale)
public java.lang.String java.lang.String.toUpperCase()
public java.lang.String java.lang.String.toUpperCase(java.util.Locale)
public java.lang.String java.lang.String.trim()
public static java.lang.String java.lang.String.valueOf(boolean)
public static java.lang.String java.lang.String.valueOf(char[],int,int)
public static java.lang.String java.lang.String.valueOf(int)
public static java.lang.String java.lang.String.valueOf(long)
public static java.lang.String java.lang.String.valueOf(float)
public static java.lang.String java.lang.String.valueOf(double)
public static java.lang.String java.lang.String.valueOf(char[])
public static java.lang.String java.lang.String.valueOf(java.lang.Object)
public static java.lang.String java.lang.String.valueOf(char)
public final native java.lang.Class java.lang.Object.getClass()
public final void java.lang.Object.wait(long,int) throws java.lang.InterruptedException
public final void java.lang.Object.wait() throws java.lang.InterruptedException
public final native void java.lang.Object.wait(long) throws java.lang.InterruptedException
public final native void java.lang.Object.notify()
public final native void java.lang.Object.notifyAll()
.....Public Fields (1).....
public static final java.util.Comparator java.lang.String.CASE_INSENSITIVE_ORDER
.....Declared Methods (70).....
public int java.lang.String.hashCode()
public int java.lang.String.compareTo(java.lang.String)
public int java.lang.String.compareTo(java.lang.Object)
public int java.lang.String.indexOf(int,int)
public int java.lang.String.indexOf(int)
```

```
public int java.lang.String.indexOf(java.lang.String)
public int java.lang.String.indexOf(java.lang.String,int)
static int java.lang.String.indexOf(char[],int,int,char[],int,int,int)
public boolean java.lang.String.equals(java.lang.Object)
public java.lang.String java.lang.String.toString()
public char java.lang.String.charAt(int)
private static void java.lang.String.checkBounds(byte[],int,int)
public int java.lang.String.codePointAt(int)
public int java.lang.String.codePointBefore(int)
public int java.lang.String.codePointCount(int,int)
public int java.lang.String.compareToIgnoreCase(java.lang.String)
public java.lang.String java.lang.String.concat(java.lang.String)
public boolean java.lang.String.contains(java.lang.CharSequence)
public boolean java.lang.String.contentEquals(java.lang.StringBuffer)
public boolean java.lang.String.contentEquals(java.lang.CharSequence)
public static java.lang.String java.lang.String.copyValueOf(char[])
public static java.lang.String java.lang.String.copyValueOf(char[],int,int)
public boolean java.lang.String.endsWith(java.lang.String)
public boolean java.lang.String.equalsIgnoreCase(java.lang.String)
public static java.lang.String java.lang.String.format(java.lang.String,java.lang.Object[])
public static java.lang.String
java.lang.String.format(java.util.Locale,java.lang.String,java.lang.Object[])
public byte[] java.lang.String.getBytes()
public byte[] java.lang.String.getBytes(java.nio.charset.Charset)
public byte[] java.lang.String.getBytes(java.lang.String) throws
java.io.UnsupportedEncodingException
public void java.lang.String.getBytes(int,int,byte[],int)
void java.lang.String.getChars(char[],int)
public void java.lang.String.getChars(int,int,char[],int)
public native java.lang.String java.lang.String.intern()
public boolean java.lang.String.isEmpty()
public int java.lang.String.lastIndexOf(int)
public int java.lang.String.lastIndexOf(java.lang.String)
```

```
public int java.lang.String.lastIndexOf(java.lang.String,int)
public int java.lang.String.lastIndexOf(int,int)
static int java.lang.String.lastIndexOf(char[],int,int,char[],int,int,int)
public int java.lang.String.length()
public boolean java.lang.String.matches(java.lang.String)
public int java.lang.String.offsetByCodePoints(int,int)
public boolean java.lang.String.regionMatches(boolean,int,java.lang.String,int,int)
public boolean java.lang.String.regionMatches(int,java.lang.String,int,int)
public java.lang.String
java.lang.String.replace(java.lang.CharSequence,java.lang.CharSequence)
public java.lang.String java.lang.String.replace(char,char)
public java.lang.String java.lang.String.replaceAll(java.lang.String,java.lang.String)
public java.lang.String java.lang.String.replaceFirst(java.lang.String,java.lang.String)
public java.lang.String[] java.lang.String.split(java.lang.String,int)
public java.lang.String[] java.lang.String.split(java.lang.String)
public boolean java.lang.String.startsWith(java.lang.String,int)
public boolean java.lang.String.startsWith(java.lang.String)
public java.lang.CharSequence java.lang.String.subSequence(int,int)
public java.lang.String java.lang.String.substring(int,int)
public java.lang.String java.lang.String.substring(int)
public char[] java.lang.String.toCharArray()
public java.lang.String java.lang.String.toLowerCase()
public java.lang.String java.lang.String.toLowerCase(java.util.Locale)
public java.lang.String java.lang.String.toUpperCase()
public java.lang.String java.lang.String.toUpperCase(java.util.Locale)
public java.lang.String java.lang.String.trim()
public static java.lang.String java.lang.String.valueOf(boolean)
public static java.lang.String java.lang.String.valueOf(char[],int,int)
public static java.lang.String java.lang.String.valueOf(int)
public static java.lang.String java.lang.String.valueOf(long)
public static java.lang.String java.lang.String.valueOf(float)
public static java.lang.String java.lang.String.valueOf(double)
public static java.lang.String java.lang.String.valueOf(char[])
```

```

public static java.lang.String java.lang.String.valueOf(java.lang.Object)
public static java.lang.String java.lang.String.valueOf(char)
.....Declared Fields (7).....
private final char[] java.lang.String.value
private final int java.lang.String.offset
private final int java.lang.String.count
private int java.lang.String.hash
private static final long java.lang.String.serialVersionUID
private static final java.io.ObjectStreamField[] java.lang.String.serialPersistentFields
public static final java.util.Comparator java.lang.String.CASE_INSENSITIVE_ORDER

```

Example 20.2:

The previous example is modified so as to read the fully qualified class name as command line argument and then display the information about the class's methods and fields using reflection/introspection.

```

1.      import java.lang.reflect.Method;
2.      import java.lang.reflect.Field;
3.      class DispClassMembers2
4.      {
5. public static void main(String args[]) throws ClassNotFoundException
6.      {
7.          Class c = Class.forName(args[0]);
8.          Method m[] = c.getMethods();
9.      System.out.println(".....Public Methods (" + m.length + ").....");
10.         for(int i=0; i< m.length; i++)
11.             System.out.println(m[i]);
12.         Field f[] = c.getFields();
13.     System.out.println(".....Public Fields (" + f.length + ").....");
14.         for(int i=0; i< f.length; i++)
15.             System.out.println(f[i]);
16.         m = c.getDeclaredMethods();
17.     System.out.println(".....Declared Methods (" + m.length + ").....");
18.         for(int i=0; i<m.length; i++)

```

```

19.                System.out.println(m[i]);
20.                f = c.getDeclaredFields();
21.    System.out.println(".....Declared Fields (" + f.length + ").....");
22.                for(int i=0; i< f.length; i++)
23.                    System.out.println(f[i]);
24.                }
25.    }

```

Output:

Same as previous but pass the class name "String" from commandline arguments.

```
javac DispClassMembers2.java
```

```
java DispClassMembers2 String
```

4. String toString():

If a sub class does not override this method, it returns a textual representation of the object, which has the following format:

"<name of the class>@<hash code value of object>"

The method is usually overridden and used for debugging purposes. The method call `System.out.println(Objref)` will implicitly convert its argument to a textual representation using `toString()` method.

Example 20.3:

This example demonstrates what gets displayed if we try to display object of class Box. the object is converted to the textual representation using the `toString` method of the object class.

```

1.    class Box
2.    {
3.        double w,h,d;
4.        Box(double w, double h, double d)
5.        {
6.            this.w=w;        this.h=h;        this.d=d;
7.        }
8.    }
9.    class ToStringDemo1
10.   {
11.       public static void main(String args[])

```

```

12.         {
13.             Box b= new Box(10,12,14);
14.             String s="Box b:"+b;
15.             System.out.println(b);
16.             System.out.println(s);
17.         }
18.     }

```

Output:

```

Box1@82b41
Box1 b: Box1@82b41

```

Example 20.4:

This example demonstrate that if we override the toString() method in the Box class then the overridden method is used to convert the object to its textual representation.

```

1.     class Box
2.     {
3.         double w,h,d;
4.         Box(double w, double h, double d)
5.         {
6.             this.w=w;         this.h=h;         this.d=d;
7.         }
8.         public String toString()
9.         {
10.            return "Dimensions are "+ w + " by "+ h + " by " + d + ".";
11.        }
12.    }
13.    class ToStringDemo2
14.    {
15.        public static void main(String args[])
16.        {
17.            Box b= new Box(8,11,13);
18.            String s="Box b:" + b;           //concatenates box object & calling toString() method
19.            System.out.println(b);           // Calling toString() method
20.            System.out.println(s);

```


21. }

22. }

Output:

Dimension are 8.0 by 11.0 by 13.0

Box b: Dimension are 8.0 by 11.0 by 13.0

5. protected void finalize() throws Throwable

It is called on an object just before it is garbage collected, so that any cleaning up can be done. However, the default finalize() method in the object class does not do anything useful. This may be useful for releasing non-java resources but not recommended. It is possible that finalize() method may never be called if enough memory is available and in that case resources may never be released.

6. protected Object clone() throws CloneNotSupportedException

New objects that are exactly the same (i.e. have identical states) as the current object can be created by the clone() method, that is, primitive values and reference values are copied, this is called **shallow cloning**.

A class can override the clone() method to provide its own notion of cloning. For example, cloning a composite object by recursively cloning the constituent object is called **deep cloning**. When overridden, the method in the subclass is usually declared public to allow any client to clone objects of the class.

If overriding clone() method relies on the clone() method in the Object class, then the subclass must implement the cloneable marker interface to indicate that its objects can be safely cloned. Otherwise, the clone() method in the Object class will throw a checked CloneNotSupportedException.

Using clone() and the Cloneable interface:

The clone() method generates a duplicate copy of the object on which it is called. These are few important facts related to clone() method:

Only classes that implement the Cloneable interface can be cloned. The Cloneable interface defines no members. It is used to indicate that a class allows a bit-wise copy of an object (that is, a clone) to be made.

If you try to call clone() on object of a class that does not implement cloneable interface, CloneNotSupportedException is thrown.

Cloneable interface is an empty interface. Such an interface is called **marker/tag interface**. When a clone is made, the constructor for the object being cloned is not called.

A clone is simply an exactly copy of the original. Cloning is potentially a dangerous action, because it can cause unintended side effects.

For example, if the object being cloned contains a reference variable called objRef, then when the clone is made the objRef in clone will refer to the same object as does objRef in original. If the clone makes a change to the contents of the object referred to by objRef, then it will be changed for the original object, too.

Example20.5:

The following example demonstrates the use of clone() method. The CloneDemo1 class is making clone of the object of class TestClone1 by indirectly calling the clone() method through clone2() method.

```
1.      class TestClone1 implements Cloneable
2.      {
3.          int a;
4.          float b;
5.      public TestClone1 clone2() throws CloneNotSupportedException
6.          {
7.      return (TestClone1) clone();
8.          }
9.      }
10.     class CloneDemo1
11.     {
12. public static void main(String a[]) throws CloneNotSupportedException
13.         {
14.             TestClone1 tc1 = new TestClone1();
15.             TestClone1 tc2;
16.             tc1.a = 8; tc1.b = 4.5f;
17.             tc2 = tc1.clone2();
18.             System.out.println(tc2.a);
19.             System.out.println(tc2.b);
20.         }
21.     }
```

Output:

```
8
4.5
```

Here, the method clone2() calls clone() of Object class and returns the result. Notice that the object returned by clone() must be cast into its appropriate type i.e. TestClone1.

Example 20.6: In the following example, the clone() method is overridden so that it can be called from code outside of its class. To do this, its access modifier must be public.

```
1.      class TestClone2 implements Cloneable
```

```

2.      {
3.          int a;
4.          float b;
5.          public Object clone() throws CloneNotSupportedException
6.          {
7.              return super.clone();
8.          }
9.      }
10.     class CloneDemo2
11.     {
12.     public static void main(String args[]) throws CloneNotSupportedException
13.     {
14.         TestClone2 tc1 = new TestClone2();
15.         TestClone2 tc2;
16.         tc1.a = 8; tc1.b = 4.5f;
17.         tc2 = (TestClone2) tc1.clone();
18.         System.out.println(tc2.a + " , " + tc2.b);
19.     }
20.     }

```

Output: 8 , 4.5

Example 20.7: In the following example, the clone() method is overridden such that it does not make use of the clone() method of the Object class. We are writing our own clone() method.

```

1.     class TestClone3
2.     {
3.         int a;
4.         float b;
5.         public Object clone()
6.         {
7.             TestClone3 tc = new TestClone3();
8.             tc.a = a;
9.             tc.b = b;
10.            return tc;

```

```

11.         }
12.     }
13.     class CloneDemo3
14.     {
15.         public static void main(String args[])
16.         {
17.             TestClone3 tc1 = new TestClone3();
18.             TestClone3 tc2;
19.             tc1.a = 8;
20.             tc1.b = 4.5f;
21.             tc2 = (TestClone3) tc1.clone();
22.             System.out.println(tc2.a + " , " + tc2.b);
23.         }
24.     }

```

Output:

8 , 4.5

Note: You need not declare CloneNotSupportedException and need not implement Cloneable interface if implementing you own clone() method.

Side Effect of Cloning-

The side effects caused by cloning are sometimes difficult to see at first. It is easy to think that a class is safe for cloning when it actually is not.

In general, you should not implement Cloneable interface for any class without good reason.

Example20.8: This example demonstrates the side effect of cloning.

```

1.     class TestShallowClone implements Cloneable
2.     {
3.         int a;
4.         float b;
5.         int c[ ];
6.         public Object clone() throws CloneNotSupportedException
7.         {
8.             return super.clone();
9.         }

```

```

10.     }
11.         class ShallowCloneDemo
12.     {
13.     public static void main(String arg[]) throws CloneNotSupportedException
14.         {
15.             TestShallowClone tc1 = new TestShallowClone();
16.             TestShallowClone tc2;
17.             tc1.a = 8;
18.             tc1.b = 4.5f;
19.                 int c[] = new int[4];
20.                 for(int i=0; i<4; i++)
21.                     c[i] = i;
22.                 tc1.c = c;
23.                 for(int i=0; i<4; i++)
24.                     System.out.println("tc1.c[" + i + "]= " + tc1.c[i]);
25.                 tc2 = (TestShallowClone) tc1.clone();
26.                 System.out.println("tc2.a"+tc2.a);
27.                 System.out.println("tc2.b"+tc2.b);
28.                 for(int i=0; i<4; i++)
29.                     System.out.println("tc2.c[" + i + "]= " + tc2.c[i]);
30.                 for(int i=0;i<4;i++)
31.                     tc2.c[i] = i+4;
32.                 for(int i=0; i<4; i++)
33.                     System.out.println("tc2.c[" + i + "]= " + tc2.c[i]);
34.                 for(int i=0; i<4; i++)
35.                     System.out.println("tc1.c[" + i + "]= " + tc1.c[i]);
36.             }
37.     }

```

Output:

```

tc1.c[0]=0
tc1.c[1]=1
tc1.c[2]=2
tc1.c[3]=3
tc2.a=8
tc2.b=4.5
tc2.c[0]=0
tc2.c[1]=1
tc2.c[2]=2

```

```
tc2.c[3]=3
tc2.c[0]=4
tc2.c[1]=5
tc2.c[2]=6
tc2.c[3]=7
tc1.c[0]=4
tc1.c[1]=5
tc1.c[2]=6
tc1.c[3]=7
```

Example 20.9:The following code eliminates the problem faced in the previous example by implementing deep cloning.

```
1. class TestDeepClone implements Cloneable
2. {
3.     int a;
4.     float b;
5.     int c[];
6.     public Object clone() throws CloneNotSupportedException
7.     {
8.         TestDeepClone tc = (TestDeepClone)super.clone();
9.         int d[] = new int[4];
10.        for(int i=0; i<4; i++)
11.            d[i] = c[i];
12.        tc.c = d;
13.        return (tc);
14.    } }
15.
16. class DeepCloneDemo
17. {
18.     public static void main(String arg[]) throws CloneNotSupportedException
19.     {
20.         TestDeepClone tc1 = new TestDeepClone();
21.         TestDeepClone tc2;
22.         tc1.a = 8;
23.         tc1.b = 4.5;
24.         int c[] = new int[4];
25.         for(int i=0; i<4; i++)
26.             c[i] = i;
27.         tc1.c = c;
28.         for(int i=0; i<4; i++)
```

```

29.         System.out.println("tc1.c["+ i +"]=" + tc1.c[i]);
30.         tc2 = (TestDeepClone) tc1.clone();
31.         System.out.println("tc2.a"+tc2.a);
32.         System.out.println("tc2.b"+tc2.b);
33.         for(int i=0; i<4; i++)
34.             System.out.println("tc2.c[" + i + "]" + tc2.c[i]);
35.         for(int i=0; i<4; i++)
36.             tc2.c[i] = i+4;
37.         for(int i=0; i<4; i++)
38.             System.out.println("tc2.c[" + i + "]" + tc2.c[i]);
39.         for(int i=0; i<4; i++)
40.             System.out.println("tc1.c[" + i + "]" + tc1.c[i]);
41.     } }

```

Output: tc1.c[0]=0

```

tc1.c[1]=1
tc1.c[2]=2
tc1.c[3]=3
tc2.a=8
tc2.b=4.5
tc2.c[0]=0
tc2.c[1]=1
tc2.c[2]=2
tc2.c[3]=3
tc2.c[0]=4
tc2.c[1]=5
tc2.c[2]=6
tc2.c[3]=7
tc1.c[0]=0
tc1.c[1]=1

```

Methods useful in multi-threaded environment:

In addition to methods discussed above, object class provides support for thread communication in synchronized code through the following methods: Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.

8. final void wait(long timeout) throws InterruptedException

9. final void wait(long timeout, long nanos) throws InterruptedException

Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object. or some other thread interrupts the

current thread, or the specified amount of time has elapsed.

10. final void notify()

Wakes up a single thread that is waiting on this object's monitor.

11. final void notifyAll()

Wakes up all thread that are waiting on this object's monitor. A thread waits on an object's monitor by calling one of the wait methods.

A thread invokes these methods on the object whose lock it holds. A thread waits for notification by another thread.



CHAPTER

∞ 21 ∞

(Collection Framework)

Introduction-

The Java Collections framework standardizes the way in which group of objects are handled by your programs. Prior to Java2, Java provided ad hoc classes such as Vector, Stack, Dictionary, Hashtable and Properties to store and manipulate groups of objects.

Although these classes were quite useful, they lacked a central unifying theme. The way that we use Vector was different from the way that we use properties class, Also, the previous ad hoc approach was not designed to be easily extensible or adaptable. Collections are an answer to these (and other) problems.

The Collection framework was designed keeping into consideration the following objectives:

- ✦ The Framework has to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hash tables) are highly efficient.
- ✦ The Framework has to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- ✦ Extending and/or adapting to a collection had been easy.

To achieve the above goals, the collection framework design the following features:

- ✦ Entire Collection framework is designed around a set of standard interfaces. The several standard implementations of these interfaces are provided. We may also implement our own collection.
- ✦ Another item created by the collections framework is the Iterator interface. An Iterator gives you a general-purpose, standardized ways of accessing the element within a collection, one at a time.
- ✦ Thus, an Iterator provides a means of enumerating the contents of a collection. Because each collection implement Iterator, the elements if any collection class can be accessed through the methods defined by Iterator. This, with only small changes, the code that cycle through a set can also be used to cycle through a list, for example.
- ✦ In addition to collections, the framework defines several map interfaces and classes. Maps store key/value pairs. Although maps are not “collection” in the proper use of the term, they are fully integrated with collections.
- ✦ In the language of the collection framework, we can obtain a collection view of map. Such a view contains the elements form the map stored in a collection. Thus, we can process the contents of a map as a collection, if you choose.
- ✦ The collection mechanism was retrofitted to some of the original classes defined by java.util package, so that they too could be integrated into the new system.
- ✦ It is important to understand that although the addition of collections altered the architecture of many of the original utility classes, it did not cause the deprecation of any. Collections simply provide a better way of doing several things.

The Collection Interfaces:

The collections framework defines several interfaces. The concrete classes simply provide different implementations of the standard interfaces. The interfaces that underpin collections are:

Collection

Enable you to work with group of objects; it is at the top of the collections hierarchy.

List

Extends collection to handle sequences (lists of objects).

Set

Extends collection to handle sets, which must contain unique elements.

SortedSet

Extends collection to handle sorted sets.

In addition to the collection interfaces, collections also use the Comparator, Iterator and RandomAccess interfaces. The comparator defines how two objects are compared; Iterator and ListIterator enumerate the objects within a collection.

By implementing RandomAccess, a list indicates that it supports efficient, random access to its elements. To provide the greatest flexibility in their use, the collection interfaces allow some methods to be optional.

The optional methods enable you to modify the contents of a collection. Collections that support these methods are called modifiable. Collections that do not allow their contents to be changed are called unmodifiable.

If an attempt is made to use one of these methods on an unmodifiable collection an UnsupportedOperationException is thrown. All the built-in collections are modifiable.

The Iterator Interface: Iterator interface defines the following methods:

Boolean	hasNext() returns true if the iteration has more elements.
Object	next() returns the next element in the iteration.
void	remove() Removes from the underlying collection the last element returned by the iterator (may not allow to do so in some circumstances as in EJB's). it is an optional operation , so it may throw UnsupportedOperationException, if iterator does not support this operation.

Iterators differ from enumerations in two ways:

- ✚ Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantic.
- ✚ Method names have been improved.

The ListIterator Interface :

This is a sub interface of Iterator.

public interface ListIterator extends Iterator

It is an iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.

Method summary:

void add(Object obj)

Inserts the specified element into the list (optional operation).

boolean hasNext()

Returns true if this list iterator has more elements when traversing the list in the forward direction.

boolean hasPrevious()

Returns true if this list iterator has more elements when traversing the list in the reverse direction.

Object next()

Return the next element in the list.

int nextIndex()

Returns the index of the element that would be returned by a subsequent call to next.

Object previous()

Return the previous element in the list.

int previousIndex()

Returns the index of the element that would be returned by a subsequent call to previous.

void remove()

Remove from the list the last element that was returned by next or previous (optional operation).

void set(Object obj)

Replaces the last element returned by next or previous with the specified element (optional operation).

RandomAccess Interface:

This is a marker interface used by List implement to indicate that they support random access. The ArrayList and Vector implement this interface.

The Comparator Interface:

Comparators are used to control the order of certain data structures(such as TreeSet or TreeMap).

Methods:

Int	compare (Object o1, Object o2) Compares its two arguments for order.
Boolean	equal (Object obj) Indicate whether some other object is "equal to" this Comparator.

The Collection Interface:

The collection interface is the foundation upon which the collections framework is built. It declares the core methods that all collections will have. Because all collections implement collection interface, familiarity with its methods is necessary for a clear understanding of the framework. Several of these methods can throw an UnsupportedOperationException. A ClassCastException is generated when one object is incompatible with another, such as when an attempt is made to add an incompatible object

to a collection.

Methods in the Collection Interface:

boolean add(Object obj)

Ensures that this collection contains the specified element (optional operation). Adds object to the invoking collection. Returns true if element obj was added to the collection. Returns false if element obj is already a member of the collection and the collection does not allow duplicate.

boolean addAll(Collection c)

Adds all of the elements in the specified collection to this collection(optional operation)

void clear()

Removes all if the element from this collection (optional operation).

boolean contain(Object o)

Return true if this collection contains the specified collection.

boolean containAll(Collection c)

Return true if this collection contains all of the elements in the specified collection.

boolean equal(Object o)

Compares the specified object with this collection for equality.

int hashCode()

Returns the hash code value for this collection.

boolean isEmpty()

Returns true if this collection contains no elements.

Iterator iterator()

Returns an iterator over the elements in this collection.

boolean remove(Object o)

Removes a single instance of the specified element from this collection, if it is present (optional operation)

boolean removeAll(Collection c)

Removes all the element that are also contained in the specified collection (optional operation)

boolean retainAll(Collection c)

Retains the element of this collection that are contained in the specified collection (optional operation).

int size()

Returns the number of elements in this collection.

Object[] toArray()

Returns an array containing all of the elements in this collection.

Two collections can be compared for equality by calling **equals()** method. The precise meaning of “equality” may differ from collection to collection.

For example, you can implement equals() so that it compares the values of elements stored in the collection. Alternatively, equals() can compare references to these elements.

The List Interface:

A list is an ordered collection also known as sequence. The List interface extends Collection and declares the behavior of a collection that stores a sequence of elements.

Elements can be inserted or accessed by their position in the list, using a zero based index. A list may contain duplicate elements. In addition to the methods defined by collection, List defines some of its own.

Methods in the List Interface:

boolean add(Object o)

Appends the specified element to the end of this list (optional operation). The method is defined in the Collection interface but as part of the List interface it should be implemented so as to always add the element at the end of the list.

void add(int index, Object element)

Inserts the specified element at the specified position in this list (optional operation). Any pre-existing elements at or beyond the point of insertion are sifted up. Thus, no elements are overwritten.

boolean addAll(Collection c)

Appends all of the element in the specified collection to the end of this list, in the order that they are returned by the specified collections iterator(optional operation). The method is defined in the Collection interface but as part of the List interface it should be implemented so as to always add the element at the end of the list.

boolean addAll(int index, Collection c)

inserts all of the elements in the specified collection into this list at the specified position (optional operation). Returns true if the invoking list changes and returns false otherwise.

boolean addAll(int index, Collection c)

Inserts all of the elements in the specified collection into this list at the specified position (optional operation). Returns true if the invoking list changes and returns false otherwise.

Object get(int index)

Returns the element at the specified position in this list.

int indexOf(Object o)

Returns the index in this list of the first occurrence of the specified element, or -1 if list does not contain the element.

int lastIndexOf(Object o)

Returns the index in this list of the last occurrence of the specified element, or -1 if list does not contain the element.

ListIterator iterator()

Returns a list iterator of elements in this list (in proper sequence).

ListIterator iterator(int index)

Returns a list iterator of elements in this list (in proper sequence), starting at the specified position in this list.

Object remove(int index)

Removes the elements at the specified position (optional operation) and returns deleted element. The resulting list is compacted. That is, the indexes of subsequent element are decremented by one.

Object set(int index, Object element)

Replaces the element at the specified position in this list with the specified element (optional operation).

List subList(int fromIndex, int toIndex)

Returns a list that includes elements from start to end-1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

Note: that several of these methods will throw an UnsupportedOperationException if the collection can not be modified, and a ClassCastException is generated, when an object is incompatible object to a collection.

Semantics of add(Object o) and addAll(Collection c) methods defined by collection interface is changed by list so that they add elements to the end of the list.

The Set interface:

The Set interface defines a set. It extends Collection and declares the behavior of a collection that does not allow duplicate elements.

Therefore, add() method return false if an attempt is made to add duplicate elements to a set. It does not define any additional methods of its own.

The SortedSet Interface:

The SortedSet interface extends Set and declares the behavior of a set sorted in ascending order.

Comparator comparator()

Returns the coparator associated with this sorted set, or null if it uses its elements natural ordering.

Object first()

Return the first element in this sorted set.

SortedSet headSet(Object toElement)

Returns a view of the portion of this sorted set whose elements are strictly less than toElement.

Object last()

Returns the last element in this sorted set.

SortedSet subSet(Object fromElement, Object toElement)

Returns a view of the portion of this sorted set whose elements range from fromElement, inclusive, to toElement, exclusive.

SortedSet tailSet(Object fromElement)

Returns a view of the portion of this sorted set whose elements are greater than or equal to fromElement.

Several methods throw a **NoSuchElementException** when no items array contained in the invoking set. A **ClassCastException** is thrown when an object is

incompatible with the elements in a set. A **NullPointerException** is thrown if an attempt is made to use a null object and null is not allowed in the set.

Collection Classes:

Some of the collection classes provide full implementation that can be used as is. Others are abstract, providing skeleton implementation that are used as starting points for creating concrete collection.

None of the collection classes are synchronized versions.

The standard collection classes are:

AbstractCollection Implements most of the Collection interface.

AbstractList Extends AbstractCollection and implements most of the List interface.

AbstractSequenceList Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.

AbstractSet Extends AbstractCollection and implements the Set interface.

LinkedList Implements a linked list by extending AbstractSequentialList.

ArrayList Implements a dynamic array by extending AbstractList. It uses random access of elements.

HashSet Extends AbstractSet for use with a hash table.

LinkedHashSet Extends HashSet to allow insertion-order iterations.

TreeSet Extends AbstractSet and implements SortedSet interface. It uses binary search tree to store elements in sorted order.

Note:- In addition to collection classes, several legacy classes, such as Vector, Stack, Hashtable, have been reengineered to support collections.

Accessing an Collection via Iterator and ListIterator:

Often, you will want to cycle through the elements in a collection. By far, the easiest way to do this is to employ an iterator, an object that implements either the Iterator or the ListIterator interface.

Iterator enables you to cycle through a collection obtaining or removing elements. ListIterator extends Iterator to allow bi-directional traversal of a list, and the modification of elements.

Before you can access a collection through an iterator, you must obtain one. Each of the collection classes provides an iterator() method that returns an iterator to the start of the collection. By using this iterator object, you can access each element in the collection, one element at a time.

For collections that implement List, you can also obtain an iterator by calling ListIterator. A list iterator gives you the ability to access the collection in either the forward or backward direction and you can modify an element. Otherwise, ListIterator is used just like Iterator.

ArrayList Class:

The ArrayList class extends AbstractList and implements the List interface. The ArrayList supports dynamic arrays that can grow as needed. In java, standard array are of a fixed length.

After arrays are created, they cannot grow or shrink, which means that we must know in advance how many elements an array will hold. But, sometimes, we don't know size of an array until run-time. To handle this situation, the collections framework defines ArrayList.

An ArrayList is a variable-length array of object references that is, an ArrayList can dynamically increase or decrease in size. Array lists are created with an initial size.

When the size is exceeded, the collection is automatically enlarged, when objects are removed, the array may shrink. Using generics feature introduced in J2SE 5, it is possible to restrict the type of elements to be added in the ArrayList.

Note:- The legacy class Vector also supports dynamic arrays.

Constructors & Methods of ArrayList class:

```
public ArrayList(int);
```

Constructs an empty list with the specified initial capacity. The capacity grows automatically as elements are added to an array list.

```
public ArrayList( );
```

Constructs an empty list with an initial capacity of ten.

public ArrayList(Collection);

Constructs a list containing the elements of the specified collection. In the order they are returned by the collection's Iterator.

public void trimToSize();

Conversely, if we want to reduce the size of the array that underlines an ArrayList object so that it is precisely as large as the number of items that it is currently holding call trimToSize() method.

public void ensureCapacity(int);

Although the capacity of an ArrayList object increases automatically as objects are stored in it, we can increase the capacity of an ArrayList object manually by calling ensureCapacity() method. By increasing its capacity once, at the start, we can prevent several reallocation later. Because reallocation are costly in terms of time, preventing unnecessary ones improves performance.

```
public int size( );
```

```
public Boolean isEmpty( );
```

```
public Object clone( );
```

public Object[] toArray();

Obtaining an array from an ArrayList when working with ArrayList, we will sometimes want to obtain an actual array that contain the contents of the list. We can do this by calling **toArray()** method.

Several reasons exist why we might want to convert a collection into an array such as:

- To obtain faster processing times for certain operations.
- To pass an array to a method that is not overload to accept a collection.
- To integrate our newer, collection-based code with legacy code that does not understand collection.

```
public Object[] toArray(Object[]);

public Object get(int);

public Object set(int, Object);

public boolean add(Object);

public void add(int, Object);

public Object remove(int);

public boolean remove(Object);

public void clear( );

public boolean addAll(Collection);

public boolean addAll(int, Collection);

public boolean contains(Object);

public int indexOf(Object);

public int lastIndexOf(Object);
```

Example 21.1

```
1. import java.util.*;
2. class ArrayListTest
3. {
4.     public static void main(String args[])
5.     {
6. ArrayList a1 = new ArrayList();//can store different type of objects capacity is 10
7.         System.out.println("Initial size of a1 : "+ a1.size());
8.         a1.add("A");
9.         a1.add("C");
10.        a1.add(new Integer(56));
```

```
11.      a1.add("E");
12.      a1.add("F");
13.      a1.add(1,"B");
14.      System.out.println("Size of a1 after additions: "+ a1.size());
15.      System.out.println("Contents of a1 : "+ a1);
16.      a1.remove("F");
17.      a1.remove(2);
18.      System.out.println("Size of a1 after deletions: "+ a1.size());
19.      System.out.println("Content of a1 "+ a1);
20.          ArrayList<String> a2 = new ArrayList<String>(20);
21.          //can add only String objects capacity is 20
22.          a2.ensureCapacity(25); //increase the capacity from 20 to 25.
23. a2.ensureCapacity(22); //capacity will not change as already more than
24.      a2.add("A");
25.      a2.add("B");
26.          a2.add("C");
27.          a2.add("D");
28.          a2.add("E");
29.          //a2.add(new Integer(5));//will not compile
30.      System.out.println("Contents of a2 : "+ a2);
31.      a2.remove("D");
32.      System.out.println(a2);
33.      a2.remove(0);
34.      System.out.println(a2);
35.      a2.clear();
36.          System.out.println(a2);
37.
38. ArrayList<Integer> a3 = new ArrayList<Integer>(); //can add only Integer objects
39.      a3.add(new Integer(1));
40.      a3.add(new Integer(2));
41.      a3.add(new Integer(3));
42.      a3.add(4); //Auto Boxing
```

```

43.      System.out.println("Contents of a3 : "+ a3);
44.      for(int i=0;i<a3.size();i++)
45.          System.out.print(a3.get(i)+ " ");
46.      System.out.println();
48.      Object b[]=a3.toArray();
49.      for(int i=0;i<b.length;i++)
50.          System.out.print((Integer)b[i]+ " ");
51.      System.out.println();
52.      System.out.println("Contents of a3 using iterator: ");
53.      Iterator itr = a3.iterator();
54.      while(itr.hasNext())
55.      {
56.          int x = ((Integer)itr.next()).intValue();
57.          System.out.print(x + " ");
58.      }
59.      System.out.println();
60.
61.          //Changing the list using ListIterator
62.          ListIterator<Integer> litr = a3.listIterator();
63.          while(litr.hasNext())
64.              {
65.                  Integer element = litr.next();
66.                  litr.set(element + 10);
67.              }
68.
69.          //Printing the list in reverse Order will work only after reaching the end.
70.          while(litr.hasPrevious())
71.              {
72.                  Integer element = litr.previous();
73.                  System.out.print(element + " ");
74.              }
75.          System.out.println();

```

```

76.         System.out.println("Contents of a3 using for each loop");
77.         for(int i : a3) //Auto Unboxing
78.             System.out.print(i + " ");
79.         System.out.println();
80.
81.         Integer ia[] = new Integer[a3.size()];
82.         a3.toArray(ia);
83.         int sum= 0;
84.         for(int i=0; i<ia.length; i++)
85.             sum = sum + ia[i].intValue();
86.         System.out.println("Sum is " + sum);
87.         a3.trimToSize();
88.     }
89. }

```

Output :

```

Initial size of a1 : 0
Size of a1 after additions: 6
Contents of a1 : [A, B, C, 56, E, F]
Size of a1 after deletions: 4
Content of a1 [A, B, 56, E]
Contents of a2 : [A, B, C, D, E]
[A, B, C, E]
[B, C, E]
[]
Contents of a3 : [1, 2, 3, 4]
1 2 3 4
1 2 3 4
Contents of a3 using iterator:
1 2 3 4
14 13 12 11
Contents of a3 using for each loop

```


11 12 13 14

Sum is 50

The LinkedList Class:

The LinkedList class extends AbstractSequentialList and implements the List interfaces. It provides a linked-list data structure.

Constructors & Methods

```
public LinkedList( );
```

Constructs an empty list.

```
public LinkedList(Collection);
```

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's Iterator.

```
public Object getFirst( );
```

```
public Object getLast( );
```

```
public Object removeFirst( );
```

```
public Object removeLast( );
```

```
public void addFirst(Object);
```

```
public void addLast(Object);
```

```
public boolean contains(Object);
```

```
public int size( );
```

```
public boolean add(Object);
```

```
public boolean remove(Object);
```

```
public boolean addAll(Collection);
```

```
public boolean addAll(int, Collection);
```

```
public void clear( );
```

```
public Object get(int);

public Object set(int, Object);

public void add(int, Object);

public Object remove(int);

public int indexOf(Object);

public int lastIndexOf(Object);

public Object element( );

public Object remove( );

public boolean removeFirstOccurrence(Object);

public boolean removeLastOccurrence(Object);

public ListIterator listIterator(int);

public Iterator descendingIterator( );

public Object clone( );

public Object[] toArray( );

public Object[] toArray(java.lang.Object[]);
```

Example 21.2

```
1. import java.util.*;
2. class LinkedListTest
3. {
4.     public static void main(String args[])
5.     {
6.         //Create a linked list.
7.         LinkedList<String> l1 = new LinkedList<String>();
```

```

8.
9.      //Add elements to the linked list.
10.     l1.add("F");
11.     l1.add("B");
12.     l1.add("D");
13.     l1.add("E");
14.     l1.add("C");
15.     l1.addLast("Z");
16.     l1.addFirst("A");
17.     l1.add(1, "A2");
18.     System.out.println("Original Contents of l1 : "+ l1);
20.     //Remove element from the linked list.
21.     l1.remove("F");
22.     l1.remove(2);
23.     System.out.println("Contents of l1 after deletion:" +l1);
25.     //Remove the first and last elements.
26.     l1.removeFirst();
27.     l1.removeLast();
28.     System.out.println("l1 after deleting first and last :"+ l1);
30.     // Get and set a value.
31.     String val = l1.get(2);
32.     l1.set(2, val + "Changed");
33.     System.out.println("l1 after change:" +l1);
35.     Iterator itr=l1.descendingIterator();
36.     while(itr.hasNext())
37.     {
38.         System.out.println(itr.next());
39.     } } }

```

Output:-Original Contents of l1 : [A, A2, F, B, D, E, C, Z]

Contents of l1 after deletion:[A, A2, D, E, C, Z]

l1 after deleting first and last :[A2, D, E, C]

l1 after change:[A2, D, EChanged, C]

C

EChanged

D

A2

Note- Because LinkedList implements the List interface, call to add(Object obj) append items to the end of the list, as do calls to addLast(). To insert items at a specific location use the add(int index, Object obj) form of add.

The HashSet Class:

The HashSet class extends AbstractSet and implements the Set interface. It creates a collection that uses a hash table for storage

The advantage of hashing is that it allows the execution time of basic operations, such as add(), contains(), remove(), and size() to remain constant even for large sets.

Constructors & Other Methods:

HashSet()

Constructs a new, empty set, the backing HashMap instance has default initial capacity(16) and load factor(0.75).

HashSet(Collection c)

Constructs a new set containing the elements in the specified collection.

HashSet(int initialCapacity)

Constructs a new, empty set, the backing HashMap instance has specified initial capacity and load factor which is 0.75.

HashSet(int initialCapacity, float fillRatio)

Constructs a new, empty set, the backing HashMap instance has specified initial capacity and the specified load factor. The fill ratio must be between 0.0 and 1.0

```
public java.util.Iterator iterator();
```

```
public int size( );
```

```
public boolean isEmpty( );
```

```
public boolean contains(java.lang.Object);
```

```
public boolean add(java.lang.Object);
```

```
public boolean remove(java.lang.Object);
```

```
public void clear( );
```

```
public java.lang.Object clone( );
```

The HashSet does not guarantee the order of its elements, because the process of hashing does not usually lend itself to the creation of sorted sets. If you need sorted storage, then another collection such as TreeSet, is a better choice.

Example 21.3

```
1.      import java.util.*;
2.      class HashSetTest
3.      {
4.          public static void main (String args[])
5.          {
6.              HashSet<String> hs = new HashSet<String>();
7.              hs.add("B");
8.              hs.add("A");
9.              hs.add("E");
10.             hs.add("C");
11.             hs.add("F");
12.             System.out.println(hs);
13.         }
14.     }
```

Output:

[E, F, A, B, C]

The elements are not stored in sorted order, and the precise output may vary.

The **LinkedHashSet** Class:

The `LinkedHashSet` class extends `HashSet` and adds no members of its own. The `LinkedHashSet` class maintains a linked list of the entries in the set, in the order in which they were inserted.

This is also the order in which they are contained in the string returned by `toString()` method when called on a `LinkedHashSet` object. To see the effect of `LinkedHashSet`, try substituting `LinkedHashSet` for `HashSet` in the preceding program. The output will be: [B, A, E, C, F] Which is the order in which the elements were inserted.

The **TreeSet** Class:

The `TreeSet` provides an implementation of the `Set` interface that uses a tree for storage. Objects are stored in sorted, ascending order. Access and retrieval times are quite fast, which makes `TreeSet` an excellent choice when storing large amounts of sorted information that must be found quickly.

Constructors

It has the following constructors:

TreeSet()

Constructs a new, empty set, sorted according to the elements natural order.

TreeSet(Collection c)

Constructs a new set containing the elements in the specified collection sorted according to the elements' natural order.

TreeSet(Comparator c)

Constructs a new, empty set, sorted according to the comparator.

TreeSet(SortedSet ss)

Constructs a new set containing the elements in the specified sorted according to the same ordering.

```
public java.util.Iterator iterator();  
public java.util.Iterator descendingIterator();  
public int size( );  
public boolean isEmpty( );  
public boolean contains(java.lang.Object);  
public boolean add(java.lang.Object);  
public boolean remove(java.lang.Object);  
public void clear( );  
public boolean addAll(java.util.Collection);  
public java.lang.Object clone();
```

Example 21.4

```
1.      import java.util.*  
2.      class TreeSetTest  
3.      {  
4.          public static void main (String args[])  
5.          {  
6.              TreeSet<String> hs = new TreeSet<String>();  
7.              ts.add("C");  
8.              ts.add("A");  
9.              ts.add("D");  
10.             ts.add("E");  
11.             ts.add("F");  
12.             ts.add("B");  
13.             System.out.println(ts);
```

```
14.         }
```

```
15.     }
```

Output: [A, B, C, D, E, F]

Both TreeSet and TreeMap store elements in sorted order. However, it is the comparator that defines precisely what “sorted order” means.

By default, these classes store their elements by using what java refers to as “natural ordering”, which is usually the ordering that we would expect. If we want to order elements in a different way, then specify a comparator object when you construct the set or map. Doing so gives us the ability to govern precisely how elements are stored within sorted collections maps.

The comparator interface defines two methods: compare() and equals () as discussed earlier. The compare () method, compares two elements for order:

int compare(T obj1, T obj2)

obj1 and obj2 are the objects to be compared.

- ✚ This method returns zero if the objects are equal.
- ✚ It returns a positive value if obj1 is greater than obj2.
- ✚ It returns a negative value if obj1 is less than obj2.

The methods can throw a ClassCastException if the type of objects are not compatible for comparison.

By overriding compare(), you can alter the way that objects are ordered.

For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison.

The equals() method, tests whether an object equal the invoking comparator.

boolean equals(Object obj)

Example 21.5

```
1.     import java.util.*;
2.     class CompTest
3.     {
4.         public static void main (String args[])
5.         {
6.             TreeSet ts = new TreeSet(new MyComp());
7.             ts.add("C");
8.             ts.add("A");
```

```

9.         ts.add("D");
10.        ts.add("E");
11.        ts.add("F");
12.        ts.add("B");
13.        // Display the elements.
14.        Iterator i= ts.iterator();
15.        while(i.hasNext())
16.            {
17.                Object o = i.next();
18.                System.out.println(o + " ");
19.            }
20.        }
21.    }
22.    class MyComp implements Comparator
23.    {
24.        public int compare(Object a, Object b)
25.        {
26.            String aStr, bStr;
27.            aStr = (String) a;
28.            bStr = (String) b;
29.            //Reverse the comparison.
30.            return bStr.compareTo(aStr);
31.        }
32.    }

```

Output: F E D C B A

Working with Maps:

A map is an object that stores associations between keys and values, or key/value pairs. Given a key, we can find its value.

Both keys and values are objects. The keys must be unique, but the value may be duplicated. Some maps can accept a null key and null values, other cannot

Map Interfaces:

Map classes based on the following interfaces:

Map Maps unique keys to values.

Map.Entry Describes an element (a key/value pair) in a map.

SortedMap Extends Map so that the keys are maintained in ascending order.

The Map Interface:

The Map interface maps unique keys to values. Given a key and value you can store the value in a Map object. After the value is stored, you can retrieve it by using its key.

Method summary:

void clear()

Removes all mappings from this map (optional operation).

boolean containsKey(Object key)

Returns true if this map contains a mapping for the specified key.

boolean containsValue(Object value)

Returns true if this map maps one or more keys to the specified value.

Set entrySet()

Returns a set that contains the entries in the map. The set contains objects of type Map.Entry. this method provides a set view of the invoking map.

boolean equals(Object o)

Returns true if o is Map and contains the entries, otherwise, returns false.

Object get(Object key)

Returns the value to which this map maps the specified key.

int hashCode()

Returns the hash code value for this map.

boolean isEmpty()

Returns true if this map contains no key-value mappings.

Set keySet()

Returns a set view of the keys contained in this map.

Object put(Object key, Object value)

Associates the specified value with the specified key in this map(optional operation), overwriting any previous value associated with the key. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.

void putAll(Map m)

Copies all of the mappings from the specified map to this map(optional operation).

Object remove(Object key)

Removes the mapping for this key from this map if it is present(optional operation).

int size()

Returns the number of key-value mappings in this map.

Collection values()

Returns a collection view of the values contained in this map.

Several methods throw a **NoSuchElementException** when no items exist in the invoking map. A **ClassCastException** is thrown when an object is incompatible with the elements in a map.

A **NullPointerException** is thrown if an attempt is made to use a null object and null is not allowed in the Map. An **UnsupportedOperationException** is thrown when an attempt is made to change an unmodifiable map.

Map resolve around two basic operations get() and put().

Maps are not collections because they do not implement the Collection interface, but we can obtain a collection-view of the map. Collection-view means by which maps are integrated into the collections framework..

The SortedMap Interface:

The SortedMap interface extends Map. It ensures that the entries are maintained in ascending order.

Method Summary:

Comparator comparator()

Returns the comparator associated with this sorted map, or null if it uses its keys' natural ordering.

Object firstKey()

Returns the first key in this sorted map.

SortedMap headMap(Object toKey)

Returns a view of the portion of this sorted map whose keys are strictly less than toKey.

Object lastKey()

Returns the last key in this sorted map.

SortedMap subMap(Object fromKey, Object toKey)

Returns a view of the portion of this sorted map whose keys are greater than equal to toKey., exclusive

SortedMap tailMap(Object fromKey)

Returns a view of the portion of this sorted map whose keys are greater than or equal to fromKey.

Several methods throw a **NoSuchElementException** when no items are in the invoking map. A **ClassCastException** is thrown when an object is incompatible with the elements in a map.

A **NullPointerException** is thrown if an attempt is made to use a null object and null is not allowed in the Map.

The Map.Entry Interface:

The Map.Entry interface enables you to work with a map entry. Recall that the entrySet() method declared by the Map interface returns a set containing the map entries. Each of these set elements is a Map.Entry object.

Method Summary:

--

boolean equals(Object o)

Compares the specified object with this entry for equality. Returns true if o is a Map.Entry object, whose key and value are equal to that of the invoking object.

Object getKey()

Returns the key corresponding to this entry.

Object getValue()

Returns the Value corresponding to this entry.

int hashCode()

Returns the hash code value for this map entry.

Object setValue(Object value)

Replaces the value corresponding to this entry with the specified value (optional operation). A ClassCastException is thrown if value is not of the correct type of the map.

A NullPointerException is thrown if value is null and the map does not permit null key. An UnsupportedOperationException is thrown if the map cannot be changed.

The Map classes:

Several classes provide implementation of the map interfaces.

AbstractMap This class provides a skeleton implementation of the map interface, to minimize the efforts required to implement this interface.

HashMap Hash table based implementation of the map interface. Extends AbstractMap class.

TreeMap Extends AbstractMap and implements SortedMap interface to use a tree.

LinkedHashMap Extends HashMap to allow insertion-order iteration.

AbstractMap is a super class for all concrete map implementations.

The HashMap class:

The HashMap class uses a hash table to implant the Map interface. This allows the execution time of basic operations, such as get() and put() to remain constant even for large sets.

Constructors:

The HashMap has the following constructors:

HashMap()

Constructs an empty HashMap with the default capacity(16) and default load factor (0.75)

HashMap(int initialCapacity)

Constructs an empty HashMap with the specified initial capacity and default load factor (0.75)

HashMap(Map m)

Construct a new HashMap with the same mapping as the specified Map.

The HashMap class extends AbstractMap and implements Map. It does not add any methods of its own. You should note that a hash map does not guarantee the order of its elements.

Example 21.6

```
1. import java.util.*;
2. class HashMapTest
3. {
4.     public static void main (String args[])
5.     {
6.         HashMap<String,Integer> hm = new HashMap<String,Integer>();
7.         hm.put("C",1000);
8.         hm.put ("C++",1000);
9.         hm.put ("Java",1500);
10.         Set s = hm.entrySet();
```

```

11.         Iterator itr = s.iterator();
12.         while(itr.hasNext())
13.         {
14.             Map.Entry me = (Map.Entry) itr.next();
15.             System.out.print(me.getKey() + ": " );
16.             System.out.println(me.getValue() );
17.         }
18.         System.out.println();
19.         int fees = ((Integer)hm.get("Java")).intValue();
20.         hm.put("Java", fees + 1000);
21.         System.out.println("New Fees of Java:"+hm.get("Java") );
22.     }
23. }

```

Output: C: 1000

C++: 1000

Java: 1500

New Fees of Java:2500

The TreeMap class:

The TreeMap class implements the Map interface by using a tree. A TreeMap provides an efficient means of storing key/value pair in sorted order, and tree map guarantees that its elements will be sorted in ascending key order.

Constructors:

The TreeMap has the following constructors:

TreeMap()

Constructs an empty map, sorted according to the key's natural order.

TreeMap(Comparator c)

Constructs a new, empty map, sorted according to the given comparator.

TreeMap (Map m)

Construct a new map with containing the same mapping as the given map, sorted according to the keys' natural order.

TreeMap(SortedMap m)

Construct a new map with containing the same mapping as the given SortedMap, sorted according to the same ordering.

TreeMap implements SortedMap and extends AbstractMap. It does not define any additional methods of its own.

Example 21.7

The following program reworks the preceding example so that it uses TreeMap. The keys will be sorted by first name. it is possible to alter this behavior by specifying a comparator when the map is created, so that keys may be sorted on last name or any other desired order.

```
1.  import java.util.*;
2.  class TreeMapTest
3.  {
4.      public static void main (String args[])
5.      {
6.          TreeMap <String,Integer> tm = new TreeMap<String,Integer>();
7.          tm.put("C",1000);
8.          tm.put("C++",1000);
9.          tm.put("Java",1500);
10.         Set<Map.Entry<String,Integer>> s = tm.entrySet();
11.         for(Map.Entry<String,Integer> me : s)
12.         {
13.             System.out.print(me.getKey() + ": ");
14.             System.out.println(me.getValue() );
15.         }
16.         System.out.println();
17.         int fees = tm.get("Java");
18.         tm.put("Java", fees + 1000);
19.         System.out.println("New fees of Java:"+tm.get("Java"));
20.     }
21. }
```

Output:

C: 1000

C++: 1000

Java: 1500

The LinkedHashMap Class:

Java 2, version 1.4 adds the LinkedHashMap class. This class extends HashMap. LinkedHashMap maintains a linked list of the entries in the map, in the order in which they were inserted.

This allows insertion-order iteration over the map. That is, when iterating a LinkedHashMap, the elements will be returned in the order in which they were inserted.

Legacy Classes and Interfaces

The original version of java.util did not include the collection framework. Instead, it defined several classes and an interface that provided an ad hoc method of storing object. With the addition of Collection by Java 2, several of the original classes were reengineered to support the Collection interfaces.

Thus, they are fully compatible with the framework. While no classes have actually been deprecated, one has been obsolete. Of course, where a collection duplicates the functionality of legacy class, you will usually want to use the collection for new code. In general, the legacy classes are supported because there is still code that uses them.

Some of the collection classes are synchronized, but all the legacy classes are synchronized. This action may be important in some situations. Of course, we can easily synchronize collections too.

The Enumeration Interface:

Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. The legacy interface has been superceded by Iterator.

Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods used by legacy classes (such as Vector and Properties), is used by several other API classes, and is recently in wide spread use in application code.

Enumeration specifies the following two methods:

Boolean hasMoreElements()

Tests if this enumeration contains more elements. This method must return true while there are still more elements to extract. And false when all the elements have been enumerated.

Object nextElement()

Returns the next element of this enumeration if this enumeration object has at least one more element to provide.

The Vector class

Vector implements a dynamic array. It is similar to ArrayList, but with two differences:

Vector is synchronized, and Vector contains many legacy methods that are not part of the collections framework.

With the advent of collections, Vector was reengineered to extend AbstractList and to implement the List interface.

This means that Vector is fully compatible with collections, and a vector can have its contents iterated by the enhanced for loop.

Constructors:

The Vector has the following constructors:

Vector()

Constructor an empty vector so that its internal data array has size 10 and its standard capacity increment is zero.

Vector(Collection c)

Constructs a vector containing the elements of the specified xollection in the order they are returned by the collection's iterator.

Vector(int initialCapacity)

Constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.

Vector(int initialCapacity, int capacityIncrement)

Constructs an empty vector with the specified initial capacity and capacity increment.

The size of extra space allocated during each reallocation is determined by the specified increment. If you do not specify an increment, the vector's size is doubled by each allocation cycle. Vector defines three protected data members:

protected int capacityIncrement

The amount by which the capacity of the vector is automatically incremented when its size becomes greater than its capacity.

protected int elementCount

The number of valid components in this Vector object.

protected Object[] elementData

The array buffer into which the components of the vector are stored.

In addition to the collection methods defined by list, Vector class defines several legacy methods.

Method Summary:

void addElement(Object obj)

Adds the specified components to the end of this vector, increasing its size by one.

int capacity()

Returns the current capacity of this vector.

Object clone()

Returns a clone of this vector.

void copyInto(Object[] anArray)

Copies the components of this vector into the specified array.

Object elementAt(int index)

Returns the component at the specified index.

Enumeration elements()

Returns an enumeration of the components of this vector.

void ensureCapacity(int minCapacity)

Increase the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minCapacity Argument.

Object firstElement()

Returns the first component (the item at index 0) of this vector.

int indexOf(Object element)

Searches for the first occurrence of the given argument, testing for equality using the equals methods.

int indexOf(Object element, int index)

Searches for the first occurrence of the given argument, beginning the search at index, and testing for equality using the equals methods.

void insertElementAt(Object obj, int index)

Inserts the specified object as a component in this vector at the specified index.

boolean isEmpty()

Tests if this vector has no components.

Object lastElement()

Returns the last components of the vector.

int lastIndexOf(Object element)

Returns the index of the last occurrence of the specified object in this vector.

int lastIndexOf(Object element, int index)

Searches backwards for the specified object, starting from the specified index, and returns an index to it.

void removeAllElements()

Removes all components from this vector and sets its size to zero.

boolean removeElement(Object obj)

Removes the first (lowest-indexed) occurrence of the argument from this vector.

void removeElementAt(int index)

Deletes the component at the specified index.

protected void removeRange(int fromIndex, int toIndex)

Removes from this list all of the elements whose index is between fromIndex inclusive and toIndex exclusive.

void setElementAt(Object obj, int index)

Sets the component at the specified index of this vector to be the specified object.

void setSize(int newSize)

Sets the size of this vector. If the new size is less than the old size, elements are lost. If the new size is larger than the old, null elements are added.

int size()

Returns the number of components in this vector.

List subList(int fromIndex, int toIndex)

Returns a view of the portion of this list between fromIndex inclusive, and toIndex exclusive.

Object[] toArray()

Returns an array containing all of the elements in this vector in the correct order.

String toArray()

Returns a string representation of this vector, containing the string representation of each element.

void trimToSize()

Sets the vector's capacity equal to the number of elements that it currently holds.

Because vector implement List, you can use a vector just like you use an ArrayList instance. You can also manipulate a vector using legacy methods.

Example21.8

```
1.     import java.util.*;
2.     class VectorTest
3.     {
4.         public static void main(String args[])
5.         {
6.             Vector<Integer> v1 = new Vector<Integer>();
7.             System.out.println("Size is " + v1.size() + " Capacity is " +
v1.capacity());
8.                 for(int i=1;i<=15;i++)
9.                     v1.addElement(i);
10.            System.out.println("Size is " + v1.size() + " Capacity is " +
v1.capacity());
11.                Iterator itr=v1.iterator();
12.                while(itr.hasNext())
13.                {
14.                    System.out.print(itr.next() + " ");
15.                }
16.            System.out.println();
17.            for(int i:v1)
18.                System.out.print(i + " ");
19.            System.out.println();
20.            Enumeration e=v1.elements();
21.            while(e.hasMoreElements())
22.            {
23.                System.out.print(e.nextElement() + " ");
24.            }
25.            System.out.println();
26.            Integer a[ ]=new Integer[v1.size()];
27.            v1.copyInto(a);
28.            for(int i=0;i<a.length;i++)
29.                System.out.print(a[i] + " ");
30.            System.out.println();
```

```
31.         System.out.println("First Element= " + v1.firstElement());
32.             System.out.println("Last Element= " + v1.lastElement());
33.         }
34.     }
```

Output:

Size is 0 Capacity is 10

Size is 15 Capacity is 20

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

First Element= 1

Last Element= 15

The Stack Class:

The stack is a subclass of Vector that implements a standard LIFO stack. The Stack only defines the default constructor, which creates an empty stack. Stack includes all methods defined by Vector, and adds several of its own.

Methods Summary:

boolean empty()

Tests if the stack is empty.

Object peek()

Returns the object at the top of this stack without removing it from the stack.

Object pop()

Removes the object at the top of this stack and returns that object as the value of this methods. An EmptyStackException is thrown if the stack is empty.

Object push(Object item)

Pushes an item onto the top of this stack.

int search(Object o)

Returns the 1-based position where an object is on this stack.

Example21.9:

The following example creates a stack, pushes several Integer objects onto it and pops them off again:

```
1.      import java.util.*;
2.      class StackTest
3.      {
4.          public static void main(String args[])
5.          {
6.              Stack<Integer> st=new Stack<Integer>();
7.              st.push(10);
8.              st.push(20);
9.              st.push(30);
10.             System.out.println("Stack is " + st);
11.             System.out.println("30 is at pos " + st.search(30));
12.             try
13.             {
14.                 Integer iobj=st.pop();
15.                 System.out.println("Popped value is " + iobj.intValue());
16.                 iobj = st.peek();
17.                 System.out.println("Peek value is " + iobj.intValue());
18.             }
19.             catch(EmptyStackException e)
20.             {
21.                 System.out.println(e);
22.             }
23.         }
```

24. }

Output:

Stack is [10, 20, 30]

30 is at pos 1

Popped value is 30

Peek value is 20

The Dictionary Class:

Dictionary is an **abstract class** that represents a key/value storage repository and operates much like Map. Given a key and value pair, you can store the value in a dictionary object once the value is stored, you can retrieve it by using its key.

Thus, like a map, dictionary can be thought of as a list of key/value pairs. Although not actually deprecated by java 2, Dictionary is classified as obsolete, because it is superceded by Map.

The abstract methods defined by Dictionary class are:

Enumeration elements()

Returns an enumeration of the values in this dictionary.

Object get(Object key)

Returns the value to which the key is mapped in this dictionary.

boolean isEmpty()

Tests if this dictionary maps no keys to value.

Enumeration keys()

Returns an enumeration of the keys in this dictionary.

Object put(Object key, Object value)

Maps the specified key to the specified value in this dictionary.

Object remove(Object key)

Removes the key (and its corresponding value) from this dictionary.

int size()

Returns the number of entries (distinct keys) in this dictionary.

The Hashtable Class:

The Hashtable was part of the original java.util and is a concrete implementation of a Dictionary. However, Java 2 reengineered Hashtable so that it also implements the Map interface.

Thus, Hashtable is now integrated into the collections framework. It is similar to HashMap, but is synchronized. Like HashMap.

Hashtable stores key/value pairs in a hash table. When using a Hashtable, you specify an object that is used as a key, and the value that you want to linked to that key.

The key is then hashed, and the resulting hash code is used as the index at which the value is stored within the table.

A hash table can only store objects that override the hashCode() and equals() methods that are defined by the Object class. Mostly string object is used as the key, which already implements both hashCode() and equals().

Constructors:

The Hashtable has the following constructors:

Hashtable()

Constructs a new, empty Hashtable with the default initial capacity(11) and load factor, which is 0.75.

Hashtable(int initialCapacity)

Constructs a new, empty Hashtable with the specified initial capacity and default load factor (0.75)

Hashtable(int initialCapacity, float loadFactor)

Constructs a new, empty Hashtable with the specified initial capacity and specified load factor.

Hashtable(Map m)

Construct a new hashtable with the same mapping as the specified Map.



Hashtable also defines the legacy methods:

void clear()

Clears this hashtable with that it contains no keys.

Object clone()

Creates a shallow copy of this hashtable.

boolean contains(Object value)

Tests if some key maps into the specified value in this hashtable.

boolean containsKey(Object key)

Tests if the specified object is a key in this hashtable.

boolean containsValue(Object value)

Returns true if this Hashtable maps one or more keys to this value.

Enumeration elements()

Returns an enumeration of the values in this hashtable.

Set entrySet()

Returns a Set view of the entries contained in this Hashtable.

boolean equals(Object o)

Compares the specified Object with this map for equality.

Object get(Object key)

Returns the value to which the specified key is mapped in this hashtable.

int hashCode()

Returns the hash code value for this Map as per the definition in the Map interface.

boolean isEmpty()

Tests if this hashtable maps no keys to value.

Enumeration keys()

Returns an enumeration of the keys in this hashtable.

Set keySet()

Returns a set view of the keys contained in this hashtable.

Object put(Object key, Object value)

Maps the specified key to the specified value in this hashtable.

void putAll(Map m)

Copies all of the mappings from the specified Map to this Hashtable these mappings will replace any mappings that this Hashtable had for any of the keys currently in the specified Map.

Protected void rehash()

Increase the size of the hash table and rehashes all of its keys.

Object remove(Object key)

Removes the key (and its corresponding value) from this hashtable.

int size()

Returns the number of entries (distinct keys) in this hashtable.

String toString()

Returns a string representation of this hashtable object in the form of a set of entries, enclosed in braces and separated by the ASCII character “,” (comma and space).

Collection values()

Returns a collection of the values contained in this Hashtable.

Example21.10:

The following example uses a Hashtable to store the name of depositors and their current balance.

```
1.      import java.util.*;
2.      class HashtableTest
3.      {
4.          public static void main(String args[])
5.          {
6.              Hashtable<String,Integer> ht=new Hashtable<String,Integer>();
7.              ht.put("C",1000);
8.              ht.put("Java",1500);
9.              ht.put("Vb",1500);
10.             //Display List using Enumeration
11.             Enumeration<String> courses;
12.             courses=ht.keys();
13.             while(courses.hasMoreElements())
14.             {
15.                 String s1=courses.nextElement();
16.                 System.out.println("Course= " + s1 + " Fees= " + ht.get(s1));
17.             }
18.             //Display List using for each
```

```

19.         Set s=ht.keySet();
20.         for(Object ob:s)
21. System.out.println("Course= " + (String)ob + " Fees= " + ht.get((String)ob));
22.         Integer fees=ht.get("Java");
23.         ht.put("Java",fees+1000);
24.         System.out.println("After Update ");
25.         //Display List using Iterator
26.         Iterator itr=s.iterator();
27.         while(itr.hasNext())
28.         {
29.             String s1=(String)itr.next();
30.             System.out.println("Course= " + s1 + "Fees= " +ht.get(s1));
31.         }
32.     } }

```

Output:

```

Course= Java Fees= 1500
Course= Vb Fees= 1500
Course= C Fees= 1000
Course= Java Fees= 1500
Course= Vb Fees= 1500
Course= C Fees= 1000
After Update
Course= JavaFees= 2500
Course= VbFees= 1500
Course= CFees= 1000

```

The Properties Class:

The Properties is a subclass of Hashtable. It is used to maintain list of values in which the key is a String and the value is also a string. The properties class is used by many other Java classes.

For example, it is the type of object returned by System getProperties() when obtaining environmental values.

Properties class defines the following instance variable:

Properties defaults

The variable holds a default property list associated with a Properties object.

Constructors:

The Properties has the following constructors:

Properties()

Creates an empty property list with no default values.

Properties(Properties defaults)

Creates an empty property list with the specified defaults.

In addition to the methods that properties inherits from Hashtable, Properties defines some more methods. Properties also contains one deprecated method save(). This was replaced by store() because save() did not handle errors correctly.

Method summary:

String getProperty(String key)

Returns the value associated with key. A null object is returned if key is neither in the list nor in the default property list.

String getProperty(String key, String defaultValue)

Returns the value associated with key. A default property is returned if key is neither in the list nor in the default property list.

void list(PrintStream out)

Prints this property list out to the specified output stream.

void list(PrintWriter out)

Prints this property list out to the specified output stream.

void load(InputStream inStream)

Reads a property list (key and element pairs) from the input stream.

Enumeration propertyNames()

Returns an enumeration of all the keys. This includes those keys found in the default property, too.

void save(OutputStream out, String comments)

Deprecated. This method does not throw an IOException if an I/O error occurs while saving the property list. The preferred way to save a properties list is via the store (OutputStream out, String comments) method.

Object setProperty(String key, String value)

Calls the Hashtable method put. Returns previous value associated with key, or null if no such association exists.

void store(OutputStream out, String comments)

Writes this property list (key and element pairs) in this properties table to the output stream in a format suitable for loading into a Properties table using the load method.

Example21.11:

The following program creates a property list in which the keys are the course names and the values are the duration of the courses.

```
1.      import java.util.*;
2.      class PropertiesTest
3.      {
4.          public static void main(String args[])
5.          {
6.              Properties def=new Properties();
7.              def.put("C","2 months");
8.              def.put("C++","3 months");
9.              Properties courses=new Properties(def);
10.             courses.put("Java","4 months");
11.             courses.put("Dot Net","6 months");
12.             System.out.println("Duration of Java is " + courses.getProperty("Java","Not Found"));
13.             System.out.println("Duration of C is " + courses.getProperty("C","Not Found"));
14.             System.out.println("Duration of VB is " +
courses.getProperty("VB","Not Found"));
15.             System.out.println("List of all Courses:-");
16.             Set s=courses.keySet();
17.             for(Object ob:s)
```

```

18.         {
19.         System.out.println("Duration of Course " + (String)obj + " is " +
                courses.getProperty((String)obj,"not found"));
20.         }
21.         Iterator itr = s.iterator();
22.         while(itr.hasNext())
23.         {
24.             String s1=(String)itr.next();
25.             System.out.println("Duration of Course " + s1 + " is " +
                courses.getProperty(s1,"Not Found"));
26.         }
27.     }
28. }

```

Output:

```

Duration of Java is 4 months
Duration of C is 2 months
Duration of VB is Not Found
List of all Courses:-
Duration of Course Java is 4 months
Duration of Course Dot Net is 6 months
Duration of Course Java is 4 months
Duration of Course Dot Net is 6 months

```

Using store() and load()

One of the most useful aspects of Properties is that the information contained in a Properties object can be easily stored to or loaded from disk with the store() and load().

At any time, you can write a Properties object to a stream or read it back. This makes property lists especially convenient for implementing simple databases.

For example, the following program use a property list to create a simple computerized telephone book that stores names and phone numbers. To find a person's number you enter his or her name.

The program uses the store() and load() methods to store and retrieve the list. When the program executes, it first tries to load the list from a file called phonebook.dat. if this file exists, the list is loaded. You can then add to the list.



CHAPTER

∞ 22 ∞

Java 8 Features for Developers **Lambdas, Functional interface,**

Java 8 is released in 18th March 2014, so it's high time to look for the Java 8 features for the developers. And no enough material is available for java 8. But here after a lots of hard work i am writing Some of the important features introduced in Java 8 that I am looking forward to are:

1. *forEach() method in Iterable interface.*
2. *default and static methods in Interfaces.*
3. *Functional Interfaces and Lambda Expressions.*
4. *Java Stream API for Bulk Data Operations on Collections.*
5. *Java Time API.*
6. *Collection API improvements.*
7. *Concurrency API improvements.*
8. *Java IO improvements.*
9. *Miscellaneous Core API improvements.*

Let's have a brief look on these Java 8 features. I will provide some code snippets for better understanding, so if you want to run programs in Java 8, you will have to setup Java 8 environment by following steps.

Download JDK8 and install it. Installation is simple like other java versions. JDK installation is required to write, compile and run the program in Java.

NOTE: Current Eclipse IDE doesn't support Java8, so you will have to download it from **efxclipse.org Eclipse for Java 8**. There are different versions for Mac OS, Windows and Linux systems with stable builds, so download the latest one for most features.

I just checked today (28-July-2014) and Eclipse Kepler 4.3.2 SR2 package can be used for Java 8. You need to download it first and then install "Java 8 support for Eclipse Kepler SR2" plugin from Eclipse Marketplace. I have tried this and it seems to be working fine.

1. forEach() method in Iterable interface-

Whenever we need to traverse through a Collection, we need to create an Iterator whose whole purpose is to iterate over and then we have business logic in a loop for each of the elements in the Collection. We might get **ConcurrentModificationException** if iterator is not used properly.

Java 8 has introduced FOREACH method in java.lang.Iterable interface so that while writing code we focus on business logic only. FOREACH method takes java.util.function.Consumer object as argument, so it helps in having our business logic at a separate location that we can reuse. Let's see forEach usage with simple example.

Java8ForEachExample.java

```
package com.journaldev.java8.foreach;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
```

```

import java.util.function.Consumer;
import java.lang.Integer;
public class Java8ForEachExample {

    public static void main(String[] args) {
        //creating sample Collection
        List<Integer> myList = new ArrayList<Integer>();
        for(int i=0; i<10; i++) myList.add(i);

        //traversing using Iterator
        Iterator<Integer> it = myList.iterator();
        while(it.hasNext()){
            Integer i = it.next();
            System.out.println("Iterator Value: "+i);
        }
        //traversing through forEach method of Iterable with anonymous class
        myList.forEach(new Consumer<Integer>() {

            public void accept(Integer t) {
                System.out.println("forEach anonymous class Value: "+t);
            }
        });
        //traversing with Consumer interface implementation
        MyConsumer action = new MyConsumer();
        myList.forEach(action);
    }
}
//Consumer implementation that can be reused class MyConsumer implements Consumer<Integer>{
    public void accept(Integer t) {
        System.out.println("Consumer impl Value: "+t);
    }
}

```

The number of lines might increase but forEach method helps in having the logic for iteration and business logic at separate place resulting in higher separation of concern and cleaner code.

2. default and static methods in Interfaces-

If you read `forEach` method details carefully, you will notice that it's defined in `Iterable` interface but we know that interfaces can't have method body. From Java 8, interfaces are enhanced to have method with implementation. We can use 'DEFAULT' and 'STATIC' keyword to create interfaces with method implementation.

For each method implementation in `Iterable` interface is:

```
default void forEach(Consumer<? super T> action) {
    Objects.requireNonNull(action);
    for (T t : this) {
        action.accept(t);
    }
}
```

Remember :

We know that Java doesn't provide **multiple inheritance in Classes** because it leads to **Diamond Problem**. So how it will be handled with interfaces now, since interfaces are now similar to abstract classes.

The solution is that compiler will throw exception in this scenario and we will have to provide implementation logic in the class implementing the interfaces.

Interface1.java

```
package com.journaldev.java8.defaultmethod;
@FunctionalInterface
public interface Interface1 {
    void method1(String str);
    default void log(String str){
        System.out.println("I1 logging::"+str);
    }
    static void print(String str){
```

```
System.out.println("Printing "+str);
}
//trying to override Object method gives compile time error as
//"A default method cannot override a method from java.lang.Object"
// default String toString(){
//     return "i1";
// }
}
```

Interface2.java

```
package com.journaldev.java8.defaultmethod;

@FunctionalInterface
public interface Interface2 {
    void method2();
    default void log(String str){
        System.out.println("I2 logging::"+str);
    }
}
```

Notice that both the interfaces have a common method log() with implementation logic.

MyClass.java

```
package com.journaldev.java8.defaultmethod;

public class MyClass implements Interface1, Interface2 {

    @Override
    public void method2() {
    }
}
```

```
@Override
```

```
public void method1(String str) {  
}
```

```
//MyClass won't compile without having it's own log() implementation
```

```
@Override
```

```
public void log(String str){  
    System.out.println("MyClass logging:"+str);  
    Interface1.print("abc");  
}  
}
```

As you can see that Interface1 has static method implementation that is used in MYCLASS.LOG() method implementation. Java 8 uses default and static methods heavily in Collection API and default methods are added so that our code remains backward compatible.

If any class in the hierarchy has a method with same signature, then default methods become irrelevant. Since any class implementing an interface already has Object as superclass, if we have equals(), hashCode() default methods in interface, it will become irrelevant. That's why for better clarity, interfaces are not allowed to have Object class default methods.

For complete details of interface changes in Java 8, please read *Java 8 interface changes* **Below**.

Java 8 Interface Changes – static methods, default methods, functional Interfaces-

One of the biggest design change in Java 8 is with the concept of interfaces. Prior to Java 7, we could have only method declarations in the interfaces. But from Java 8, we can have **default methods** and **static methods** in the interfaces.

Designing interfaces have always been a tough job because if we want to add additional methods in the interfaces, it will require change in all the implementing classes.

As interface grows old, the number of classes implementing it might grow to an extent that it's not possible to extend interfaces. That's why when designing an application, most of the frameworks provide a base implementation class and then we extend it and override methods that are applicable for our application. Let's look into the default and static interface methods and the reasoning of their introduction.

Interface Default Method-

For creating a default method in the interface, we need to use “**default**” keyword with the

method signature. For example,

Interface1.java

```
package com.journaldev.java8.defaultmethod;
public interface Interface1 {
    void method1(String str);
    default void log(String str){
        System.out.println("I1 logging::"+str);
        print(str);
    }
}
```

Notice that `log(String str)` is the default method in the `Interface1`. Now when a class will implement `Interface1`, it is not mandatory to provide implementation for default methods. This feature will help us in extending interfaces with additional methods, all we need is to provide a default implementation. Let's say we have another interface with following methods:

Interface2.java

```
package com.journaldev.java8.defaultmethod;
public interface Interface2 {
    void method2();
    default void log(String str){
        System.out.println("I2 logging::"+str);
    }
}
```

We know that Java doesn't allow us to extend multiple classes because it will result in the "Diamond Problem" where compiler can't decide which superclass method to use. With the default methods, the diamond problem would arise for interfaces too.

Because if a class is implementing both `Interface1` and `Interface2` and doesn't

implement the common default method, compiler can't decide which one to chose.

Extending multiple interfaces are an integral part of Java, you will find it in the core java classes as well as in most of the enterprise application and frameworks. So to make sure, this problem won't occur in interfaces, it's made mandatory to provide implementation for common default methods.

So if a class is implementing both the above interfaces, it will have to provide implementation for log() method otherwise compiler will throw error. A simple class that is implementing both Interface1 and Interface2 will be:

MyClass.java

```
package com.journaldev.java8.defaultmethod;
public class MyClass implements Interface1, Interface2 {
    @Override
    public void method2() {
    }
    @Override
    public void method1(String str) {
    }
    @Override
    public void log(String str){
        System.out.println("MyClass logging::"+str);
        Interface1.print("abc");
    }
}
```

Important points about interface default methods:

Default methods will help us in extending interfaces without having the fear of breaking implementation classes.

Default methods has bridge down the differences between interfaces and abstract classes.

Default methods will help us in avoiding utility classes, such as all the Collections class method can be provided in the interfaces itself.

Default methods will help us in removing base implementation classes, we can provide default implementation and the implementation classes can choose which one to override.

One of the major reason for introducing default methods is to enhance the Collections API in Java 8 to support lambda expressions.

If any class in the hierarchy has a method with same signature, then default methods become irrelevant. A default method cannot override a method from `java.lang.Object`.

The reasoning is very simple, it's because `Object` is the base class for all the java classes. So even if we have `Object` class methods defined as default methods in interfaces, it will be useless because `Object` class method will always be used. That's why to avoid confusion, we can't have default methods that are overriding `Object` class methods.

Default methods are also referred to as Defender Methods or Virtual extension methods.

Interface static methods -

Static methods are similar to default methods except that we can't override them in the implementation classes. This feature helps us in avoiding undesired results in case of poor implementation in child classes. Let's look into this with a simple example.

MyData.java

```
package com.journaldev.java8.staticmethod;
public interface MyData {

    default void print(String str) {
        if (!isNull(str))
            System.out.println("MyData Print::" + str);
    }

    static boolean isNull(String str) {
        System.out.println("Interface Null Check");
        return str == null ? true : "".equals(str) ? true : false;
    }
}
```

Now let's see an implementation class that is having `isNull()` method with poor implementation.

MyDataImpl.java

```
package com.journaldev.java8.staticmethod;

public class MyDataImpl implements MyData {
    public boolean isNull(String str) {
        System.out.println("Impl Null Check");
        return str == null ? true : false;
    }
    public static void main(String args[]){
        MyDataImpl obj = new MyDataImpl();
        obj.print("");
        obj.isNull("abc");
    }
}
```

Note that `isNull(String str)` is a simple class method, it's not overriding the interface method.

For example, if we will add **@Override annotation** to the `isNull()` method, it will result in compiler error. Now when we will run the application, we get following output.

```
Interface Null Check
Impl Null Check
```

If we make the interface method from static to default, we will get following output.

```
1  Impl Null Check
2  MyData Print::
3  Impl Null Check
```

The static methods are visible to interface methods only, if we remove the

isNull() method from the MyDataImpl class, we won't be able to use it for the MyDataImpl object. However like other static methods, we can use interface static methods using class name. For example, a valid statement will be:

```
1 boolean result = MyData.isNull("abc");
```

Important points about interface static methods:

Interface static methods are part of interface, we can't use it for implementation class objects.

Interface static methods are good for providing utility methods, for example null check, collection sorting etc.

Interface static method helps us in providing security by not allowing implementation classes to override them.

We can't define static methods for Object class methods, we will get compiler error as "This static method cannot hide the instance method from Object". This is because it's not allowed in java, since Object is the base class for all the classes and we can't have one class level static method and another instance method with same signature.

We can use static interface methods to remove utility classes such as Collections and move all of it's static methods to the corresponding interface, that would be easy to find and use.

Functional Interfaces -

Before I conclude the topic, I would like to provide a brief introduction to Functional interfaces. An interface with exactly one abstract method is known as Functional Interface.

A new annotation @FunctionalInterface has been introduced to mark an interface as Functional Interface. @FunctionalInterface annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces. It's optional but good practice to use it.

Functional interfaces are long awaited and much sought out feature of Java 8 because it enables us to use **lambda expressions** to instantiate them. A new package java.util.function with bunch of functional interfaces are added to provide target types for lambda expressions and method references.

3. Functional Interfaces and Lambda Expressions -

If you notice above interfaces code, you will notice @FunctionalInterface **annotation**. Functional interfaces are new concept introduced in

Java 8. An interface with exactly one abstract method becomes Functional Interface.

We don't need to use `@FunctionalInterface` annotation to mark an interface as Functional Interface. `@FunctionalInterface` annotation is a facility to avoid accidental addition of abstract methods in the functional interfaces.

You can think of it like **@Override annotation** and it's best practice to use it. `java.lang.Runnable` with single abstract method `run()` is a great example of functional interface.

One of the major benefits of functional interface is the possibility to use **lambda expressions** to instantiate them. We can instantiate an interface with **anonymous class** but the code looks bulky.

```
Runnable r = new Runnable(){
    @Override
    public void run() {
        System.out.println("My Runnable");
    }
};
```

Since functional interfaces have only one method, lambda expressions can easily provide the method implementation.

We just need to provide method arguments and business logic. For example, we can write above implementation using lambda expression as:

```
Runnable r1 = () -> {
    System.out.println("My Runnable");
};
```

If you have single statement in method implementation, we don't need curly braces also. For example above `Interface1` anonymous class can be instantiated using lambda as follows:

```
Interface1 i1 = (s) -> System.out.println(s);
i1.method1("abc");
```

So lambda expressions are means to create anonymous classes of functional interfaces easily. There are no runtime benefits of using lambda expressions, so I will use it cautiously because I don't mind writing few extra lines of code.

A new package `java.util.function` has been added with bunch of functional interfaces to provide target types for lambda expressions and method references.

Lambda expressions are a huge topic, I will write a separate + (Note) on that in future (Next Edition of this Book). You can read complete tutorial below **Java 8 Lambda Expressions Tutorial**.

Java 8 Lambda Expressions and Functional Interfaces Tutorial-

Java has always been an **Object Oriented Programming** language. What is means that everything in java programming revolves around Objects (except some primitive types for simplicity). We don't have only functions in java, they are part of Class and we need to use the class/object to invoke any function.

If we look into some other programming languages such as C++, JavaScript; they are called **functional programming language** because we can write functions and use them when required. Some of these languages support Object Oriented Programming as well as Functional Programming.

Being object oriented is not bad, but it brings a lot of verbosity to the program. For example, let's say we have to create an instance of `Runnable`. Usually we do it using anonymous classes like below.

```
1 Runnable r = new Runnable(){
2     @Override
3     public void run() {
4         System.out.println("My Runnable");
5     }
};
```

If you look at the above code, the actual part that is of use is the code inside `run()` method. Rest all of the code is because of the way java programs are structured.

Java 8 brings us the concept of **Functional Interfaces** and **Lambda Expressions** to avoid writing all the useless code that we can easily avoid by making our java compiler intelligent.

Functional Interface -

An interface with exactly one abstract method is called Functional Interface. `@FunctionalInterface` annotation is added so that we can mark an interface as functional interface. It is not mandatory to use it, but it's best practice to use it with functional interfaces to avoid addition of extra methods accidentally. If the interface is annotated with `@FunctionalInterface` annotation and we try to have more than one abstract method, it throws compiler error.

The major benefit of functional interface is that we can use **lambda expressions** to instantiate them and avoid using bulky anonymous class implementation.

Java 8 Collections API has rewritten and new Stream API is provided that uses a lot of functional interfaces. Java 8 has defined a lot of functional interfaces in `java.util.function` package, some of the useful ones are `Consumer`, `Supplier`, `Function` and `Predicate`. You can find more detail about them in **Java 8 Stream Example**.

`java.lang.Runnable` is a great example of functional interface with single abstract method `run()`.

Below code snippet provides some guidance for functional interfaces:

```
interface Foo { boolean equals(Object obj); }
// Not functional because equals is already an implicit
member (Object class)
interface Comparator<T> {
    boolean equals(Object obj);
    int compare(T o1, T o2);
}
// Functional because Comparator has only one abstract
non-Object method
interface Foo {
    int m();
    Object clone();
}
// Not functional because method Object.clone is not
public
interface X { int m(Iterable<String> arg); }
interface Y { int m(Iterable<String> arg); }
interface Z extends X, Y {}
// Functional: two methods, but they have the same
signature
interface X { Iterable m(Iterable<String> arg); }
interface Y { Iterable<String> m(Iterable arg); }
interface Z extends X, Y {}
```

```

// Functional: Y.m is a subsignature & return-type-
substitutable
interface X { int m(Iterable<String> arg); }
interface Y { int m(Iterable<Integer> arg); }
interface Z extends X, Y {}

// Not functional: No method has a subsignature of all
abstract methods
interface X { int m(Iterable<String> arg, Class c); }
interface Y { int m(Iterable arg, Class<?> c); }
interface Z extends X, Y {}

// Not functional: No method has a subsignature of all
abstract methods
interface X { long m(); }
interface Y { int m(); }
interface Z extends X, Y {}

// Compiler error: no method is return type substitutable
interface Foo<T> { void m(T arg); }
interface Bar<T> { void m(T arg); }
interface FooBar<X, Y> extends Foo<X>, Bar<Y> {}

// Compiler error: different signatures, same erasure

```

Lambda Expressions -

Lambda Expressions are the way through which we can visualize **functional programming** in the java object oriented world. Objects are the base of java programming language and we can never have a function without an Object, that's why Java language provide support for using lambda expressions only with functional interfaces.

Since there is only one abstract function in the functional interfaces, there is no confusion in applying the lambda expression to the method. Lambda Expressions syntax is **(argument) -> (body)**. Now let's see how we can write above anonymous Runnable using lambda expression.

1	Runnable r1 = () -> System.out.println("My Runnable");
---	--

Let's try to understand what is happening in the lambda expression above.

Runnable is a functional interface, that's why we can use lambda expression

to create it's instance.

Since run() method takes no argument, our lambda expression also have no argument.

Just like if-else blocks, we can avoid curly braces ({}) since we have a single statement in the method body. For multiple statements, we would have to use curly braces like any other methods.

Why do we need Lambda Expressions -

1. Reduced Lines of Code –

One of the clear benefit of using lambda expression is that the amount of code is reduced, we have already seen that how easily we can create instance of a functional interface using lambda expression rather than using anonymous class.

2. Sequential and Parallel Execution Support -

Another benefit of using lambda expression is that we can benefit from the Stream API sequential and parallel operations support. To explain this, let's take a simple example where we need to write a method to test if a number passed is prime number or not.

Traditionally we would write it's code like below. The code is not fully optimized but good for example purpose, so bear with me on this.

```
1 //Traditional approach
2 private static boolean isPrime(int number) {
3     if(number < 2) return false;
4     for(int i=2; i<number; i++){
5         if(number % i == 0) return false;
6     }
7     return true;
8 }
```

The problem with above code is that it's sequential in nature, if the number is very huge then it will take significant amount of time. Another problem with code is that there are so many exit points and it's not readable. Let's see how we can write the same method using lambda expressions and stream API.

```
1 //Declarative approach
2 private static boolean isPrime(int number) {
3     return number > 1
```



```
4    && IntStream.range(2, number - 1).noneMatch(  
5    index -> number % index == 0);  
6    }
```

IntStream is a sequence of primitive int-valued elements supporting sequential and parallel aggregate operations. This is the int primitive specialization of Stream. For more readability, we can also write the method like below.

```
1    private static boolean isPrime(int number) {  
2        IntPredicate isDivisible = index -> number % index == 0;  
3        return number > 1  
4        && IntStream.range(2, number - 1).noneMatch(  
5        isDivisible);  
6    }  
7
```

If you are not familiar with IntStream, its range() method returns a sequential ordered IntStream from startInclusive (inclusive) to endExclusive (exclusive) by an incremental step of 1. noneMatch() method returns whether no elements of this stream match the provided predicate.

It may not evaluate the predicate on all elements if not necessary for determining the result.

3. Passing Behaviors into methods -

Let's see how we can use lambda expressions to pass behavior of a method with a simple example. Let's say we have to write a method to sum the numbers in a list if they match a given criteria. We can use Predicate and write a method like below.

```
1    public static int sumWithCondition(List<Integer> numbers, Predicate<Integer>  
2    predicate) {  
3        return numbers.parallelStream()  
4        .filter(predicate)  
5        .mapToInt(i -> i)  
6        .sum();  
7
```

```
6 | }
```

Sample usage:

```
1 //sum of all numbers
2 sumWithCondition(numbers, n -> true)
3 //sum of all even numbers
4 sumWithCondition(numbers, i -> i%2==0)
5 //sum of all numbers greater than 5
6 sumWithCondition(numbers, i -> i>5)
```

4. Higher Efficiency with Laziness -

One more advantage of using lambda expression is the lazy evaluation, for example let's say we need to write a method to find out the maximum odd number in the range 3 to 11 and return square of it. Usually we will write code for this method like this:

```
1 private static int findSquareOfMaxOdd(List<Integer> numbers) {
2     int max = 0;
3     for (int i : numbers) {
4         if (i % 2 != 0 && i > 3 && i < 11 && i > max) {
5             max = i;
6         }
7     }
8     return max * max;
9 }
```

Above program will always run in sequential order but we can use Stream API to achieve this and get benefit of Laziness-seeking. Let's see how we can rewrite this code in functional programming way using Stream API and lambda expressions.

```
public static int findSquareOfMaxOdd(List<Integer> numbers) {
```

```

return numbers.stream()
.filter(NumberTest::isOdd)    //Predicate is functional interface and
.filter(NumberTest::isGreaterThan3) // we are using lambdas to initialize it
.filter(NumberTest::isLessThan11) // rather than anonymous inner classes
.max(Comparator.naturalOrder())
.map(i -> i * i)
.get();
}

public static boolean isOdd(int i) {
return i % 2 != 0;
}

public static boolean isGreaterThan3(int i){
return i > 3;
}

public static boolean isLessThan11(int i){
return i < 11;
}
}

```

If you are surprised with the double colon (::) operator, it's introduced in Java 8 and used for **method references**. Java Compiler takes care of mapping the arguments to the called method. It's short form of lambda expressions `i -> isGreaterThan3(i)` or `i -> NumberTest.isGreaterThan3(i)`.

Lambda Expression Examples -

Below I am providing some code snippets for lambda expressions with small comments explaining them.

1	
2	<code>() -> {}</code> // No parameters; void result
3	<code>() -> 42</code> // No parameters, expression body
4	<code>() -> null</code> // No parameters, expression body
5	<code>() -> { return 42; }</code> // No parameters, block body with return
6	<code>() -> { System.gc(); }</code> // No parameters, void block body
7	

```

8 // Complex block body with multiple returns
9 () -> {
10     if (true) return 10;
11     else {
12         int result = 15;
13         for (int i = 1; i < 10; i++)
14             result *= i;
15         return result;
16     }
17 }
18 (int x) -> x+1 // Single declared-type argument
19 (int x) -> { return x+1; } // same as above
20 (x) -> x+1 // Single inferred-type argument, same as below
21 x -> x+1 // Parenthesis optional for single inferred-type case
22 (String s) -> s.length() // Single declared-type argument
23 (Thread t) -> { t.start(); } // Single declared-type argument
24 s -> s.length() // Single inferred-type argument
25 t -> { t.start(); } // Single inferred-type argument
26 (int x, int y) -> x+y // Multiple declared-type parameters
27 (x,y) -> x+y // Multiple inferred-type parameters
28 (x, final y) -> x+y // Illegal: can't modify inferred-type parameters
29 (x, int y) -> x+y // Illegal: can't mix inferred and declared types
30

```

Method and Constructor References -

A method reference is used to refer to a method without invoking it; a constructor reference is similarly used to refer to a constructor without creating a new instance of the named class or array type.

Examples of method and constructor references:

```

1 System::getProperty
2 System.out::println
3 "abc"::length

```

```
4 ArrayList::new
```

```
5 int[]::new
```

That's all for Functional Interfaces and Lambda Expression Tutorial, I would strongly suggest to look into using it because this syntax is new to Java and it will take some time to grasp it and use it in a better way.

4. Java Stream API for Bulk Data Operations on Collections -

A new `java.util.stream` has been added in Java 8 to perform filter/map/reduce like operations with the collection. Stream API will allow sequential as well as parallel execution.

This is one of the best feature for me because I work a lot with Collections and usually with Big Data, we need to filter out them based on some conditions. Collection interface has been extended with `STREAM()` and `PARALLELSTREAM()` default methods to get the Stream for sequential and parallel execution. Let's see their usage with simple example.



StreamExample.java

```
package com.journaldev.java8.stream;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class StreamExample {
    public static void main(String[] args) {

        List<Integer> myList = new ArrayList<>();
        for(int i=0; i<100; i++) myList.add(i);
        //sequential stream
        Stream<Integer> sequentialStream = myList.stream();
```

```
//parallel stream
Stream<Integer> parallelStream = myList.parallelStream();

//using lambda with Stream API, filter example
Stream<Integer> highNums = parallelStream.filter(p -> p > 90);
//using lambda in forEach
highNums.forEach(p -> System.out.println("High Nums parallel="+p));

Stream<Integer> highNumsSeq = sequentialStream.filter(p -> p > 90);
highNumsSeq.forEach(p -> System.out.println("High Nums sequential="+p));
}
}
```

If you will run above example code, you will get output like this:

```
High Nums parallel=91
High Nums parallel=96
High Nums parallel=93
High Nums parallel=98
High Nums parallel=94
High Nums parallel=95
High Nums parallel=97
High Nums parallel=92
High Nums parallel=99
High Nums sequential=91
High Nums sequential=92
High Nums sequential=93
High Nums sequential=94
High Nums sequential=95
High Nums sequential=96
High Nums sequential=97
High Nums sequential=98
```

Notice that parallel processing values are not in order, so parallel processing will be very helpful while working with huge collections. Covering everything about Stream API is not possible in this topic, you can read everything about Stream API at **Java 8 Stream API Tutorial in next chapter**.



CHAPTER

∞ 23 ∞

Java 8 Stream And Time API

Java 8 Stream API Example Tutorial -

In the last topics , we looked into **Java 8 Interface Changes** and **Functional Interfaces and Lambda Expressions**.

Now we will look into one of the major API introduced in Java 8 – **Java Stream API**.

1. ***Stream API Overview.***
2. ***Collections and Streams.***
3. ***Commonly used Functional Interfaces in Stream.***
 1. ***Function and BiFunction.***
 2. ***Predicate and BiPredicate.***
 3. ***Consumer and BiConsumer.***
 4. ***Supplier.***
4. ***java.util.Optional.***
5. ***java.util.Spliterator.***
6. ***Intermediate and Terminal Operations.***
7. ***Short Circuiting Operations.***
8. ***Java Stream Examples.***
 1. ***Creating Streams.***

2. ***Converting Stream to Collection or Array.***
 3. ***Stream Intermediate Operations.***
 4. ***Stream Terminal Operations.***
9. ***Java Stream API Limitations.***

Stream API Overview -

Before we look into Java 8 Stream API Examples, let's see why it was required. Suppose we want to iterate over a list of integers and find out sum of all the integers greater than 10. Prior to Java 8, the approach to do it would be:

```
1 private static int sumIterator(List<Integer> list) {
2     Iterator<Integer> it = list.iterator();
3     int sum = 0;
4     while (it.hasNext()) {
5         int num = it.next();
6         if (num > 10) {
7             sum += num;
8         }
9     }
10    return sum;
11 }
```

There are three major problems with the above approach:

- 1) We just want to know the sum of integers but we would also have to provide how the iteration will take place, this is also called **external iteration** because client program is handling the algorithm to iterate over the list.
- 2) The program is sequential in nature, there is no way we can do this in parallel easily.
- 3) There is a lot of code to do even a simple task.

To overcome all the above shortcomings, Java 8 introduces Stream API. We can use Stream API to implement **internal iteration**, that is better because java framework is in control of the iteration. **Internal iteration** provides several features such as sequential and parallel execution, filtering based on the given criteria, mapping etc.

Most of the Stream API method arguments are functional interfaces, so lambda expressions work very well with them. Let's see how can we write above logic in a single line statement.

```
1 private static int sumStream(List<Integer> list) {
2     return list.stream().filter(i -> i > 10).mapToInt(i -> i).sum();
3 }
```

Notice that above program utilizes java framework iteration strategy, filtering and mapping methods and would increase efficiency. First of all we will look into the core concepts of Stream API and then we will go through some examples for understanding most commonly used methods.

Collections and Streams-

A collection is an in-memory data structure to hold values and before we start using collection, all the values should have been populated. Whereas a Stream is a data structure that is computed on-demand.

Stream doesn't store data, it operates on the source data structure (collection and array) and produce pipelined data that we can use and perform specific operations. Such as we can create a stream from the list and filter it based on a condition.

Stream operations use functional interfaces, that makes it a very good fit for functional programming using lambda expressions. As you can see in the above example that using lambda expressions make our code readable and short.

Stream internal iteration principle helps in achieving lazy-seeking in some of the stream operations. For example filtering, mapping, or duplicate removal can be implemented lazily, allowing higher performance and scope for optimization.

Streams are consumable, so there is no way to create a reference to stream for future usage. Since the data is on-demand, it's not possible to reuse the same stream multiple times.

Stream support sequential as well as parallel processing, parallel processing can be very helpful in achieving high performance for large collections.

All the Stream API interfaces and classes are in the java.util.stream package. Since we can use primitive data types such as int, long in the collections using auto-boxing and these operations could take a lot of time, there are specific classes for these – IntStream, LongStream and DoubleStream.

Commonly used Functional Interfaces in Stream-

Some of the commonly used functional interfaces in the Stream API methods are:

1. Function and BiFunction:

Function represents a function that takes one type of argument and returns another type of argument. Function is the generic form where T is the type of the input to the function and R is the type of the result of the function. For handling primitive types, there are specific Function interfaces-

ToIntFunction, ToLongFunction, ToDoubleFunction, ToIntBiFunction,

ToLongBiFunction, ToDoubleBiFunction, LongToIntFunction, LongToDoubleFunction, IntToLongFunction, IntToDoubleFunction etc.

Some of the Stream methods where Function or its primitive specialization is used are:

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

IntStream mapToInt(ToIntFunction<? super T> mapper) – similarly for long and double returning primitive specific stream.

IntStream flatMapToInt(Function<? super T, ? extends IntStream> mapper) – similarly for long and double

```
<A> A[] toArray(IntFunction<A[]> generator)
```

```
<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator,  
BinaryOperator<U> combiner)
```

2. Predicate and BiPredicate:

It represents a predicate against which elements of the stream are tested. This is used to filter elements from the stream. Just like Function, there are primitive specific interfaces for int, long and double.

Some of the Stream methods- where Predicate or BiPredicate specializations are used are:

```
Stream<T> filter(Predicate<? super T> predicate)
```

```
boolean anyMatch(Predicate<? super T> predicate)
```

```
boolean allMatch(Predicate<? super T> predicate)
```

```
boolean noneMatch(Predicate<? super T> predicate)
```

3. Consumer and BiConsumer:

It represents an operation that accepts a single input argument and returns no result. It can be used to perform some action on all the elements of the stream. Some of the Stream methods where Consumer, BiConsumer or its primitive specialization interfaces are used are:

```
Stream<T> peek(Consumer<? super T> action)
```

```
void forEach(Consumer<? super T> action)
```

```
void forEachOrdered(Consumer<? super T> action)
```

4. Supplier: Supplier represent an operation through which we can generate new values in the stream. Some of the methods in Stream that takes Supplier argument are:

```
public static<T> Stream<T> generate(Supplier<T> s)
```

```
<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T>  
accumulator, BiConsumer<R, R> combiner)
```

java.util.Optional -

Optional is a container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value. Stream terminal

operations return Optional object. Some of these methods are:

- Optional<T> reduce(BinaryOperator<T> accumulator)
- Optional<T> min(Comparator<? super T> comparator)
- Optional<T> max(Comparator<? super T> comparator)
- Optional<T> findFirst()
- Optional<T> findAny()

java.util.Spliterator-

For supporting parallel execution in Stream API, Spliterator interface is used. Spliterator trySplit method returns a new Spliterator that manages a subset of the elements of the original Spliterator.

Intermediate and Terminal Operations-

Stream API operations that returns a new Stream are called intermediate operations. Most of the times, these operations are lazy in nature, so they start producing new stream elements and send it to the next operation. Intermediate operations are never the final result producing operations.

Commonly used intermediate operations are filter and map. Stream API operations that returns a result or produce a side effect. Once the terminal method is called on a stream, it consumes the stream and after that we can't use stream. Terminal operations are eager in nature i.e they process all the elements in the stream before returning the result.

Commonly used terminal methods are forEach, toArray, min, max, findFirst, anyMatch, allMatch etc. You can identify terminal methods from the return type, they will never return a Stream.

Short Circuiting Operations-

An intermediate operation is called short circuiting, if it may produce finite stream for an infinite stream. For example limit() and skip() are two short circuiting intermediate operations.

A terminal operation is called short circuiting, if it may terminate in finite time for infinite stream. For example anyMatch, allMatch, noneMatch, findFirst and findAny are short circuiting terminal operations.

Java Stream Examples-

I have covered almost all the important parts of the Java Stream API. It's exciting to use this new API features and let's see it in action with some examples.

Creating Streams-

There are several ways through which we can create a stream from array and collections. Let's look into these with simple examples.

1. We can use Stream.of() to create a stream from similar type of data. For example, we

can create Stream of integers from a group of int or Integer objects.

```
1 Stream<Integer> stream = Stream.of(1,2,3,4);
```

2. We can use Stream.of() with an array of Objects to return the stream. Note that it doesn't support autoboxing, so we can't pass primitive type array.

```
1 Stream<Integer> stream = Stream.of(new Integer[]{1,2,3,4});
2 //works fine
3
4 Stream<Integer> stream1 = Stream.of(new int[]{1,2,3,4});
5 //Compile time error, Type mismatch: cannot convert from Stream<int[]> to
   Stream<Integer>
```

3. We can use Collection stream() to create sequential stream and parallelStream() to create parallel stream.

```
1 List<Integer> myList = new ArrayList<>();
2 for(int i=0; i<100; i++) myList.add(i);
3 //sequential stream
4 Stream<Integer> sequentialStream = myList.stream();
5 //parallel stream
6 Stream<Integer> parallelStream = myList.parallelStream();
```

4. We can use Stream.generate() and Stream.iterate() methods to create Stream.

```
1 Stream<String> stream1 = Stream.generate(() -> {return "abc";});
2 Stream<String> stream2 = Stream.iterate("abc", (i) -> i);
```

5. Using Arrays.stream() and String.chars() methods.

```
1 LongStream is = Arrays.stream(new long[]{1,2,3,4});
2 IntStream is2 = "abc".chars();
```

Converting Stream to Collection or Array-

There are several ways through which we can get a Collection or Array from a Stream.

1. We can use Stream collect() method to get List, Map or Set from stream.

```
1 Stream<Integer> intStream = Stream.of(1,2,3,4);
2 List<Integer> intList = intStream.collect(Collectors.toList());
3 System.out.println(intList); //prints [1, 2, 3, 4]
4 intStream = Stream.of(1,2,3,4); //stream is closed, so we need to create it again
5 Map<Integer,Integer> intMap = intStream.collect(Collectors.toMap(i -> i, i -> i+10));
6 System.out.println(intMap); //prints {1=11, 2=12, 3=13, 4=14}
```

2. We can use stream toArray() method to create an array from the stream.

```
1 Stream<Integer> intStream = Stream.of(1,2,3,4);
2 Integer[] intArray = intStream.toArray(Integer[]::new);
3 System.out.println(Arrays.toString(intArray)); //prints [1, 2, 3, 4]
```

Stream Intermediate Operations- Let's look into commonly used Stream intermediate operations example.

1. Stream filter() example: We can use filter() method to test stream elements for a condition and generate filtered list.

```
1 List<Integer> myList = new ArrayList<>();
2 for(int i=0; i<100; i++) myList.add(i);
3 Stream<Integer> sequentialStream = myList.stream();
4 Stream<Integer> highNums = sequentialStream.filter(p -> p > 90); //filter numbers
greater than 90
5 System.out.print("High Nums greater than 90=");
6 highNums.forEach(p -> System.out.print(p+" "));
```

```
7 //prints "High Nums greater than 90=91 92 93 94 95 96 97 98 99 "
```

2. Stream map() example: We can use map() to apply functions to an stream. Let's see how we can use it to apply upper case function to a list of Strings.

```
1 Stream<String> names = Stream.of("aBc", "d", "ef");
2 System.out.println(names.map(s -> {
3     return s.toUpperCase();
4     }).collect(Collectors.toList()));
5 //prints [ABC, D, EF]
```

3. Stream sorted() example: We can use sorted() to sort the stream elements by passing Comparator argument.

```
1 Stream<String> names2 = Stream.of("aBc", "d", "ef", "123456");
2 List<String> reverseSorted =
3 names2.sorted(Comparator.reverseOrder()).collect(Collectors.toList());
4 System.out.println(reverseSorted); // [ef, d, aBc, 123456]
5 Stream<String> names3 = Stream.of("aBc", "d", "ef", "123456");
6 List<String> naturalSorted = names3.sorted().collect(Collectors.toList());
7 System.out.println(naturalSorted); //[123456, aBc, d, ef]
```

4. Stream flatMap() example: We can use flatMap() to create a stream from the stream of list. Let's see a simple example to clear this doubt.

```
1 Stream<List<String>> namesOriginalList = Stream.of(
2     Arrays.asList("Pankaj"),
3     Arrays.asList("David", "Lisa"),
4     Arrays.asList("Amit"));
5 //flat the stream from List<String> to String stream
6 Stream<String> flatStream = namesOriginalList
```

```
7     .flatMap(strList -> strList.stream());
8     flatStream.forEach(System.out::println);
```

Stream Terminal Operations-

Let's look at some of the terminal operations example.

1. Stream reduce() example: We can use reduce() to perform a reduction on the elements of the stream, using an associative accumulation function, and return an Optional. Let's see how we can use it multiply the integers in a stream.

```
1     Stream<Integer> numbers = Stream.of(1,2,3,4,5);
2     Optional<Integer> intOptional = numbers.reduce((i,j) -> {return i*j;});
3     if(intOptional.isPresent()) System.out.println("Multiplication = "+intOptional.get());
    //120
```

2. Stream count() example: We can use this terminal operation to count the number of items in the stream.

```
1     Stream<Integer> numbers1 = Stream.of(1,2,3,4,5);
2     System.out.println("Number of elements in stream="+numbers1.count()); //5
```

3. Stream forEach() example: This can be used for iterating over the stream. We can use this in place of iterator. Let's see how to use it for printing all the elements of the stream.

```
1     Stream<Integer> numbers2 = Stream.of(1,2,3,4,5);
2     numbers2.forEach(i -> System.out.print(i+", ")); //1,2,3,4,5,
```

4. Stream match() examples: Let's see some of the examples for matching methods in Stream API.

```
1     Stream<Integer> numbers3 = Stream.of(1,2,3,4,5);
2     System.out.println("Stream contains 4? "+numbers3.anyMatch(i -> i==4));
3     //Stream contains 4? true
4     Stream<Integer> numbers4 = Stream.of(1,2,3,4,5);
5     System.out.println("Stream contains all elements less than 10? "+numbers4.allMatch(
6     i<10));
```

```
7 //Stream contains all elements less than 10? true
8 Stream<Integer> numbers5 = Stream.of(1,2,3,4,5);
9 System.out.println("Stream doesn't contain 10? "+numbers5.noneMatch(i -> i==10));
10 //Stream doesn't contain 10? true
```

5. Stream findFirst() example: This is a short circuiting terminal operation, let's see how we can use it to find the first string from a stream starting with D.

```
1 Stream<String> names4 = Stream.of("Pankaj","Amit","David", "Lisa");
2 Optional<String> firstNameWithD = names4.filter(i -> i.startsWith("D")).findFirst();
3 if(firstNameWithD.isPresent()){
4     System.out.println("First Name starting with D="+firstNameWithD.get()); //David
5 }
```

Java Stream API Limitations -

Stream API brings a lot of new stuffs to work with list and arrays, but it has some limitations too.

1. Stateless lambda expressions: If you are using parallel stream and lambda expressions are stateful, it can result in random responses. Let's see it with a simple program.

StatefulParallelStream.java

```
1 package com.journaldev.java8.stream;
2 import java.util.ArrayList;
3 import java.util.Arrays;
4 import java.util.List;
5 import java.util.stream.Stream;
6 public class StatefulParallelStream {
7     public static void main(String[] args) {
8         List<Integer> ss = Arrays.asList(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15);
9         List<Integer> result = new ArrayList<Integer>();
10        Stream<Integer> stream = ss.parallelStream();
11        stream.map(s -> {
12            synchronized (result) {
13                if (result.size() < 10) {
```



```
14 | result.add(s);
15 | }
16 | }
17 | return s;
18 | }).forEach( e -> {});
19 | System.out.println(result);
20 | } }
```

If we run above program, you will get different results because it depends on the way stream is getting iterated and we don't have any order defined for parallel processing. If we use sequential stream, then this problem will not arise.

2. Once a Stream is consumed, it can't be used later on. As you can see in above examples that every time I am creating a stream.

3. There are a lot of methods in Stream API and the most confusing part is the overloaded methods. It makes the learning curve time taking.

That's all for Stream API in Java. I am looking forward to use this feature and make the code readable with better performance through parallel processing.

5. Java Time API -

It has always been hard to work with Date, Time and Time Zones in java. There was no standard approach or API in java for date and time in Java. One of the nice addition in Java 8 is the java.time package that will streamline the process of working with time in java.

Just by looking at Java Time API packages, I can sense that it will be very easy to use. It has some sub-packages java.time.format that provides classes to print and parse dates and times and java.time.zone provides support for time-zones and their rules.

The new Time API prefers enums over integer constants for months and days of the week. One of the useful class is DateTimeFormatter for converting datetime objects to strings.

6. Collection API improvements -

We have already seen forEach() method and Stream API for collections. Some new methods added in Collection API are:

Iterator default method for- EachRemaining(Consumer action) to perform the given action for each remaining element until all elements have been processed or the action throws an exception.

Collection default method- removeIf(Predicate filter) to remove all of the elements of this collection that satisfy the given predicate.

Collection spliterator() method- returning Spliterator instance that can be

used to traverse elements sequentially or parallel.

Map `replaceAll()`, `compute()`, `merge()` methods.

7. *Concurrency API improvements* -

Some important concurrent API enhancements are:

- a) `ConcurrentHashMap` `compute()`, `forEach()`, `forEachEntry()`, `forEachKey()`, `forEachValue()`, `merge()`, `reduce()` and `search()` methods.
- b) `CompletableFuture` that may be explicitly completed (setting its value and status).
- c) Executors `newWorkStealingPool()` method to create a work-stealing thread pool using all available processors as its target parallelism level.

8. *Java IO improvements* -

Some IO improvements known to me are:

- d) `Files.list(Path dir)` that returns a lazily populated Stream, the elements of which are the entries in the directory.
- e) `Files.lines(Path path)` that reads all lines from a file as a Stream.
- f) `Files.find()` that returns a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.
- g) `BufferedReader.lines()` that return a Stream, the elements of which are lines read from this `BufferedReader`.

9. *Miscellaneous Core API improvements* -

Some misc API improvements that might come handy are:

- h) **ThreadLocal** static method `withInitial(Supplier supplier)` to create instance easily.
- i) **Comparator** interface has been extended with a lot of default and static methods for natural ordering, reverse order etc.
- j) `min()`, `max()` and `sum()` methods in `Integer`, `Long` and `Double` wrapper classes.
- k) `logicalAnd()`, `logicalOr()` and `logicalXor()` methods in `Boolean` class.
- l) **ZipFile**.`stream()` method to get an ordered Stream over the ZIP file entries. Entries appear in the Stream in the order they appear in the central directory of the ZIP file.
- m) Several utility methods in `Math` class.

That's all for major improvements in Java 8.

*Both Physical Paperback and Digital Editions Are Available on **LuLu.com** &*

Amazon.com (Paperback Editions)
Google Books & Google Play Book Stores (PDF)

Order today and Get a Discounted Copy.

Join me on Facebook- <https://www.facebook.com/harry.novelist>
<https://www.facebook.com/CaptainHarrychoudhary?ref=hl>



CHAPTER

∞ 24 ∞

Key Features that Make Java More Secure than Other Languages.

Today, Java is driving more than \$100 billion of business annually. If we take a look at the enterprise side, more than \$2.2 billion are being spent by the enterprises in Java application server.

There is no denying that Java is used extensively for developing Java enterprise applications reason being Security. Java brings some of the most fascinating features or benefits that are impossible to find in any other programming languages or platforms.



Security is an important aspect and Java's security model is one of the key architectural features that make it most trustful choice when it comes to developing enterprise-level applications. Security becomes critical when software is downloaded across network and executed locally, and Java easily mitigates the security vulnerabilities associated with the projects or applications. Don't believe this? Have a look at a few arguments (security measures/features) that showcase how secure Java platform is.

Java's security model

Java's security model is intended to help and protect users from hostile programs downloaded from some un-trusted resource within a network through "sandbox". It allows all the Java programs to run inside the sandbox only and prevents many activities from un-trusted resources including reading or writing to the local disk, creating any new process or even loading any new dynamic library while calling a native method.

No use of pointers

C/C++ language uses pointers, which may cause unauthorized access to memory blocks when other programs get the pointer values. Unlike conventional C/C++ language, Java never uses any kind of pointers. Java has its internal mechanism for memory management. It only gives access to the data to the program if has appropriate verified authorization.

Exception handling concept

The concept of exception handling enables Java to capture a series of errors that helps developers to get rid of risk of crashing the system.

Defined order execution

All the primitives are defined with a predefined size and all the operations are defined in a specific order of execution. Therefore, the code executed in different Java Virtual

Machines won't have a different order of execution.

Byte code is another thing that makes Java more secure

Every time when a user compiles the Java program, the Java compiler creates a class file with Bytecode, which are tested by the JVM at the time of program execution for viruses and other malicious files.

Tested code re-usability

The Java object encapsulation provides support for the concept of "programming by contract". This allows the developers to re-use the code that has already been tested while developing Java enterprise applications.

Access Control functionality

Java's access-control functionality on variables and methods within the objects provide secure program by preventing access to the critical objects from the un-trusted code.

Protection from security attacks

It allows developers to declare classes or methods as FINAL. We all know that any class or method declared as final can't be overridden, which helps developers to protect code from security attacks like creating a subclass and replacing it with the original class and override methods.

Garbage collection mechanism

Garbage collection mechanism aids more to the security measures of Java. It provides a transparent storage allocation and recovering unutilized memory rather than de-allocating memory through manual action. It will help developers to ensure the integrity of the program during its execution and avoids any JVM crash due to incorrect freeing of memory.

Type-safe reference casting in JVM

Whenever you use an object reference, the JVM monitors you. If you try to cast a reference to a different type, it will make the cast invalid.

Apart from all these, structured error handling contributes a lot to the security model of Java by helping to enhance the robustness of the programs. The above arguments definitely prove that the projects developed in Java are more secure as compared to the other programming language. However, it is the responsibility of the developers to follow some best practices while developing enterprise-level Java applications.