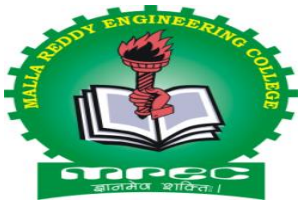
Department of Computer Science and Engineering (CS,AIML,DS,IOT)

Material



II B. Tech I Semester

Subject: Object Oriented Programming Through Java

Code: A0510

Academic Year 2021-2022
Regulations: MR20

2020-21 Onwards (MR-20)	MALLA REDDY ENGINEERING COLLEGE (Autonomous)	B.Tech. III Semester		
Code: A0510	**JAVA PROGRAMMING** (Common for CSE and IT)	**L**	**T**	**P**
Credits: 3		**3**	**-**	**-**

Prerequisites: Computer Programming

Course Objectives:
This course will make students able to learn and understand the concepts and features of object oriented programming and the object oriented concept like inheritance and will know how to make use of interfaces and package, to acquire the knowledge in Java's exception handling mechanism, multithreading, to explore concepts of Applets and event handling mechanism. This course makes students to gain the knowledge in programming using Layout Manager and swings.

MODULE I: OOP concepts & Introduction to C++, Java [09 Periods]
OOP concepts & Introduction to C++ - Introduction to object oriented concepts : Object, class, methods, instance variables; C++ program structure; Standard Libraries; accessing class data members; Overview of Inheritance, Overloading, Polymorphism, Abstraction, Encapsulation and Interfaces.
Introduction to Java - History of JAVA, Java buzzwords, data types, variables, scope and life time of variable, arrays, operators, expressions, control statements ,type conversion and type casting, simple Java program.

MODULE II: Basics of JAVA [09 Periods]
Classes and Objects - Concepts of classes, Objects, constructors, methods, this key word , garbage collection overloading methods, constructors parameter passing ,recursion.String handling: string, string buffer, string tokenizer.
Inheritance - Base class object, subclass, member access rules, super uses, using final with inheritance, method overriding, abstract classes

MODULE III: Interfaces and Exception Handling [12 Periods]
A: Interfaces - Defining an interface, implementing interface, differences between classes and interfaces, extending interfaces. Packages - Defining, creating and accessing a package, importing packages, access control, exploring package-java.io.
B: Exception handling - Concepts of Exception handling, benefits of exception handling, exception hierarchy, checked and unchecked exceptions, usage of try, catch , throw, throws and finally, built-in exceptions, creating own exception subclasses.

MODULE IV: Multithreading and Collection Classes [09 Periods]
Multithreading - Differences between multithreading and multitasking, thread life cycle, creating threads, synchronizing threads, daemon threads, thread groups.

Collection Classes –Array List, LinkedList, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDequeue, EnumSet.

MODULE V: Event handling, Layout manager and Swings [09Periods]
Event handling - Events, Event sources, event classes, event listeners, delegation event model, handling mouse and keyboard events, adapter classes. Layout manager - border, grid, flow, card and grid bag.**Layout manager -** Layout manager types-border, grid, flow, card, and grid bag.

Swings - Introduction, limitations of AWT, components, containers, exploring swing-JApplet, JFrame and JComponent,Icons and Labels, TextFields, buttons – the JButton class, Checkboxes, Radio buttons, Combo boxes, Tabbed Panes, ScrollPanes, Trees and Tables.

TEXTBOOKS

1. Herbert Schildt, "**Java The complete reference**", TMH, 8th edition
2. T. Budd, "**Understanding OOP with Java**", updated edition, Pearson Education.
3. Joyce Farrell, Cengage , "**Object Oriented Programming C++"**, 4th Edition ,2013

REFERENCES

1. P.J. Deitel and H.M. Deitel, "**Java for Programmers**", Pearson education.
2. P. Radha Krishna, "**Object Orientd Programming through Java**", Universities Press.
3. S. Malhotra and S. Choudhary, "**Programming in Java**", Oxford Univ. Press.
4. Bruce Eckel, "**Programming in Java**", Pearson Education.
5. Herbert Schildt , "**The Complete Reference, C++"**, TMH, 4th edition.

E-RESOURCES

1. http://ndl.iitkgp.ac.in/document/xttk-4kfhvUwVlXBW-WRPf64_TFk2i4LJhgQFPQWAEt-Zobbm3twyubjRA1YOe9WVwkN2qGcxBwdHaPdi_mMQ
2. https://ndl.iitkgp.ac.in/result?q={"t":"search","k":"object%20oriented%20programming","s":["type=\"video\""],"b":{"filters":[]}}
3. http://www.rehancodes.com/files/oop-using-c++-by-joyce-farrell.pdf
4. http://www.nptel.ac.in/courses/106103115/36

Course Outcomes:

At the end of the course, students will be able to

1. **Differentiate** structured programming and object-oriented programming and know the concepts of classes, objects, members of a class.
2. **Apply** object-oriented programming features and concepts for solving given problems using inheritance and will know how to organize files in packages and concept of interface.
3. **Capable** of handling run time errors using Exceptional Handling and develop applications for concurrent processing using Thread Concept.
4. **Design** Applets that take user response through various peripheral devices such as mouse and keyboard by event handling mechanism.
5. **Design** interactive applications for use on internet.

PEO's Mapping with PO's

Program Educational Objective	Program Outcomes(a-i)								
	a	b	c	d	e	f	g	h	i
PEO I	✓	✓	✓	✓	✓		✓	✓	✓
PEO II	✓	✓		✓	✓	✓	✓		
PEO III				✓	✓	✓	✓		
PEO IV				✓		✓	✓	✓	✓

Subject Mapping with PEO's

The components of the curriculum and their relevance to the POs and the PEOs

Programme curriculum grouping based on different components

Course Component	Curriculum Content (%of number of credits of the programme)	Total no of contact hours	Total number of credits	POs	PEOs
Professional core	**II Year I Semester**				
	Java Programming	**48**	**3**	**a,b,c,f,g,h**	**P1,P2**

Subject Mapping with PO's

Core engineering subjects and their relevance to Programme Outcomes including design experience.

Contribution of courses to program outcomes	**Program Outcomes (a-i)**									**Course outcomes**
II YEAR I SEMSTER										
Course No. & Title	a	b	c	d	e	f	g	h	i	
Java Programming	✓	✓	✓	✓	✓	✓	✓			**Differentiate** the structured programming & object-oriented programming, know the concepts of classes, objects, members of a class.

Year/ Sem	**Course Name**	**Outcomes**	**Program Outcomes**	**Highly**	**Moderately**
II/I	**Java Programming**	**Differentiate** structured programming and object oriented programming and know the concepts of classes, objects, members of a class.	a,b,c,d,e,f	b,c	d,e
		Apply OOP features and concepts for solving given problems using inheritance and will know how to organize files in packages & concept of interface.	b,c,d,e	B	c,d,e
		Capable of handling run time errors using Exceptional Handling and develop applications for concurrent processing using Thread Concept.	a,c,d,f,g	a,c,d	f,g
		Design Applets that take user response through various peripheral devices by event handling mechanism.	b,c,e	b	c,e
		Design interactive applications for use on internet.	b,c,d,e	b,e	c,d

Course Objectives and Course Outcomes

Course Objectives:

This course will make students able to learn and understand the concepts and features of object-oriented programming and the object-oriented concept like inheritance and will know how to make use of interfaces and package, to acquire the knowledge in Java's exception handling mechanism, multithreading, to explore concepts of Applets and event handling mechanism. This course makes students to gain the knowledge in programming using Layout Manager and swings.

Course Outcomes:

At the end of the course, students will be able to

1. **Differentiate** structured programming and object-oriented programming and know the concepts of classes, objects, members of a class.
2. **Apply** object-oriented programming features and concepts for solving given problems using inheritance and will know how to organize files in packages and concept of interface.
3. **Capable** of handling run time errors using Exceptional Handling and develop applications for concurrent processing using Thread Concept.
4. **Design** Applets that take user response through various peripheral devices such as mouse and keyboard by event handling mechanism.
5. **Design** interactive applications for use on internet.

Module-1

Syllabus:

OOP concepts & Introduction to C++: Introduction to object-oriented concepts: Object, class, methods, Instance variables; C++ program structure; accessing class data members; Overview of Inheritance, Overloading,Polymorphism, Abstraction, Encapsulation.

Introduction to Java – History of JAVA,Java buzzwords,Data types, variables,scope and life time of variable,arrays, operators, expressions, control statements, type conversion and type casting, arrays, simple Java program.

Introduction to object-oriented concepts

Class:

- A class is a template from which objects are created. That is objects are instance of a class.
- When you create a class, you are creating a **new datatype**. You can use this type to declare objects of that type.
- Class defines structure and behavior (data & code) that will be shared by a set of objects.

Object:

- An object is a region of storage that defines both **state**&**behavior**.
- **State** is represented by a set of variables & the values they contain.
- **Behavior** is represented by a set of methods & the logic they implement.
- Thus, an object is a combination of a data & the code that acts upon it.
- Objects are instance of a class.
- Objects are the basic runtime entities in an object-oriented system.

For Example:

```
Person Ram, Ravi;
Ram = new person();
Ravi = new person();
```

Method:

- A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the reusability of code. We write a method once and use it many times.
- It is similar to a function in any other programming language.

Why Use Methods?

1. To Make the Code Reusable
2. To Simplify the Code.
3. For Top-down Programming

Method Type:

- There are two types of methods in Java: *instance methods* and *class methods.*
- Instance methods are used to access/manipulate the instance variables but can also access class variables.
- Class methods can access/manipulate class variables but cannot access the instance variables unless and until they use an object for that purpose.

Method Declaration:

- The name and parameter list for each method in a class must be unique.
- The uniqueness of a parameter is decided based on the number of parameters as well as the order of the parameters.
- Let us take a look at the general syntax of a method declaration:

```
[modifiers] return_type  method_name (parameter_list) [throws_clause] {
            [statement_list]
}
```

- The parameters enclosed within square brackets [ ] are optional. The method declaration includes

modifiers.

- We have the following list of modifiers. They are public, private, protected, or default, static, abstract, final, native, synchronized etc.
- Return Type It can be either void (if no value is returned) or if a value is returned, it can be either a primitive type or a class. If the method declares a return type, then before it exits, it must have a return statement.
- Method Name The method name must be a valid Java identifier.
- Parameter List Zero or more type/identifier pairs make up a parameter list. Each parameter in the parameter list is separated by a comma.
  Curly Braces The method body is contained in a set of curly braces. Methods contain a sequence of statements that are executed sequentially. The method body can also be empty.

What is instance variable in Java?

Instance variables in Java are non-static variables which are defined in a class outside any method, constructor or a block. Each instantiated object of the class has a separate copy or instance of that variable. An instance variable belongs to a class.

When you create a new object of the class you create an instance. Consider, if you have a STUDENT class, then

```
class Student
{
String studentName;
int studentScore;
}
```

Features of an instance variable?

The life of an instance variable depends on the life of an Object, i.e., when the object is created, an instance variable also gets created and the same happens when an object is destroyed.

- Instance Variable can be used only by creating objects
- Every object will have its own copy of Instance variables
- Initialization of instance variable is not compulsory. The default value is zero
- The declaration is done in a class outside any method, constructor or block
- Instance variables are used when the variable has to be known to different methods in a class
- Access modifiers can be assigned to instance variables

Structure of a C++ program

A **C++ program** is structured in a specific and particular manner. In C++, a program is divided into the following three sections:

1. Standard Libraries Section
2. Main Function Section
3. Function Body Section

Example:

```
#include <iostream>
using namespace std;
```

```
int main() {
  cout << "Hello World!" << endl;
  return 0;
}
```

Output:

Hello World!

Standard library section:

- #include is a specific preprocessor command that effectively copies and pastes the entire text of the file, specified between the angle brackets, into the source code.
- The file <iostream>, which is a standard file that should come with the C++ compiler, is short for **input-output streams**. This command contains code for displaying and getting an input from the user.
- namespace is a prefix that is applied to all the names in a certain set. iostream file defines two *names* used in this program - **cout** and **endl**.
- This code is saying: Use the cout and endl tools from the std toolbox.

Main function section:

- The starting point of all C++ programs is the main function.
- This function is called by the operating system when your program is executed by the computer.
- { signifies the start of a block of code, and } signifies the end.

Function body section:

- The name cout is short for **character output** and displays whatever is between the << brackets.
- Symbols such as << can also behave like functions and are used with the keyword cout.
- The return keyword tells the program to return a value to the function int main
- After the return statement, execution control returns to the operating system component that launched this program.
- Execution of the code terminates here.

To access the members of a class

1. First of all, import the class.
2. Create an object of that class.
3. Using this object access, the members of that class.

OOP Principles

All object-oriented programming languages provide mechanism that help you implement the object-oriented model. They are:

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

Encapsulation:

- Encapsulation is the mechanism that binds code and data together and keeps both safe from outside interference and misuse.
- In Java, the basis of encapsulation is the class.
- When we create a class, we will specify the code and data that constitute that class. Collectively, these elements are called members of the class.
- The data defined by the class are referred to as member variables or instance variables.
- The code that operates on that data is referred to as member methods or just method.
- Each method or variable in a class may be marked private or public.
- The public interface of a class represents everything that external users of the class need to know or may know.
- The private methods and data can only be accessed by code that is a member of the class. Therefore, any other code that is not a member of the class cannot access a private method or variable.

Abstraction:

- Elephant, CD player, Television, Chair, Table , Tiger from these objects we can specify we have 3 classes. They are animals, furniture and electronic items.
- We got this solution because we grouped the generic characteristics rather than specific characteristics. This is called abstraction.
- Another well-known analogy for abstraction is a car. We drive cars without knowing the internal details about how the engine works and how the car stops on applying brakes.
- We are happy with the abstraction provided to us, e.g., brakes, steering, etc. and we interact with them.

Inheritance:

- The concept of Inheritance provides the idea of reusability.
- This means that we can add additional features to an existing class without modifying it. This is possible by deriving new class form the existing one.
- The new class have the combined feature of both the classes.
- Each subclass defines only those features that are unique to it.
- Derived class is known as ‘sub class’ and main class is known as ‘super class’.

Polymorphism:

- Polymorphism means the ability to take more than one form.
- A single function name can be used to handle different no and different types of argument.
- It plays an important role in allowing objects having different internal structures to share the same external interface.

History of Java

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as Green Team), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. Following are given significant points that describe the history of Java.

1) **James Gosling, Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.

2) Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.

3) Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.

4) After that, it was called **Oak** and was developed as a part of the Green project.

Why Java was named as "Oak"?

5) **Why Oak?** Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.

6) In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies.

Why Java Programming named "Java"?

7) Why had they chose the name Java for Java language? The team gathered to choose a new name. The suggested words were "dynamic", "revolutionary", "Silk", "jolt", "DNA", etc. They wanted something that reflected the essence of the technology: revolutionary, dynamic, lively, cool, unique, and easy to spell, and fun to say.

According to James Gosling, "Java was one of the top choices along with **Silk**". Since Java was so unique, most of the team members preferred Java than other names.

8) Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.

9) Notice that Java is just a name, not an acronym.

10) Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

11) In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.

12) JDK 1.0 was released on January 23, 1996. After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds new features in Java.

Java Version History

Many java versions have been released till now. The current stable release of Java is Java SE 10.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan 1996)
3. JDK 1.1 (19th Feb 1997)
4. J2SE 1.2 (8th Dec 1998)
5. J2SE 1.3 (8th May 2000)
6. J2SE 1.4 (6th Feb 2002)
7. J2SE 5.0 (30th Sep 2004)
8. Java SE 6 (11th Dec 2006)
9. Java SE 7 (28th July 2011)
10. Java SE 8 (18th Mar 2014)
11. Java SE 9 (21st Sep 2017)
12. Java SE 10 (20th Mar 2018)
13. Java SE 11 (September 2018)
14. Java SE 12 (March 2019)
15. Java SE 13 (September 2019)
16. Java SE 14 (Mar 2020)
17. Java SE 15 (September 2020)
18. Java SE 16 (Mar 2021)
19. Java SE 17 (September 2021)
20. Java SE 18 (to be released by March 2022

Java Buzzwords (or)Features of Java

The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

1. Simple
2. Object-oriented
3. Distributed
4. Interpreted
5. Robust
6. Secure
7. Architecture neutral
8. Portable
9. High performance
10. Multithreaded
11. Dynamic

1. Simple

- Java was designed to be easy for a professional programmer to learn and use effectively.
- It's simple and easy to learn if you already know the basic concepts of Object Oriented Programming.
- Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java.
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.

2. Object Oriented

- Java is true object-oriented language.
- Almost "Everything is an Object" paradigm. All program code and data reside within objects and classes.
- The object model in Java is simple and easy to extend.
- Java comes with an extensive set of classes, arranged in packages that can be used in our programs through inheritance.
- Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

- Object
- Class
- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

3. Distributed

- Java is designed fa or distributed environment of the Internet. Its used for creating applications on networks.
- Java applications can access remote objects on the Internet as easily as they can do in the local system.
- Java enables multiple programmers at multiple remote locations to collaborate and work together on a single project.
- Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols.

4. Compiled and Interpreted

- Usually, a computer language is either compiled or Interpreted. Java combines both this approach and makes it a two-stage system.
- Compiled: Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java Bytecode.
- Interpreted: Bytecode is then interpreted, which generates machine code that can be directly executed by the machine that provides a Java Virtual machine.

5. Robust

- It provides many features that make the program execute reliably in a variety of environments.
- Java is a strictly typed language. It checks code both at compile time and runtime.
- Java takes care of all memory management problems with garbage collection.
- Java, with the help of an exception handling, captures all types of serious errors and eliminates any risk of crashing the system.

6. Secure

- Java provides a “firewall” between a networked application and your computer.
- When a Java Compatible Web browser is used, downloading can be done safely without fear of viral infection or malicious intent.
- Java achieves this protection by confining a Java program to the Java execution environment and not allowing it to access other parts of the computer.

7. Architecture Neutral

- Java language and Java Virtual Machine helped in achieving the goal of “write once; run anywhere, any time, forever.”
- Changes and upgrades in operating systems, processors and system resources will not force any changes in Java Programs.

8. Portable

- Java Provides a way to download programs dynamically to all the various types of platforms connected to the Internet.
- It helps in generating Portable executable code.

9. High Performance

- Java performance is high because of the use of bytecode.
- The bytecode was used so that it was easily translated into native machine code.

10. Multithreaded

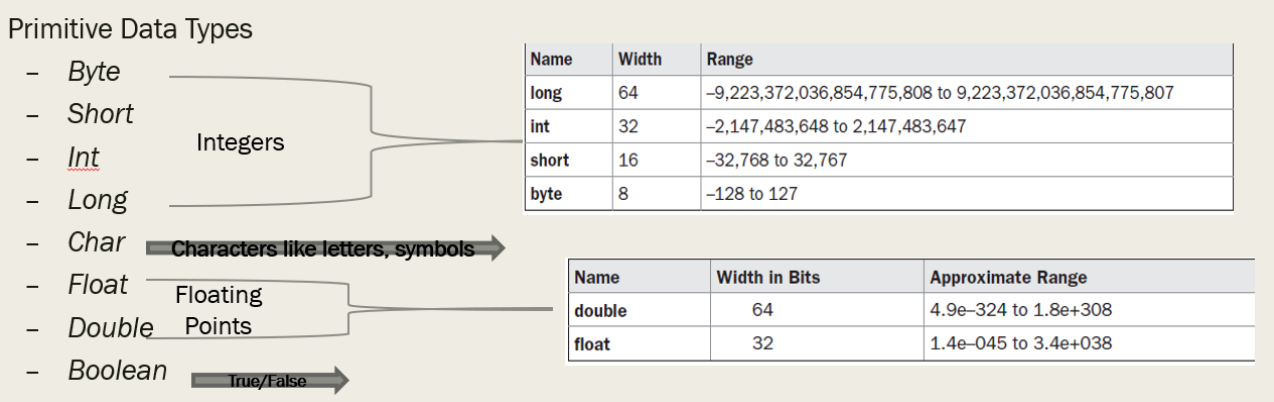
- Multithreaded Programs handled multiple tasks simultaneously, which was helpful in creating interactive, networked programs.
- Java run-time system comes with tools that support multiprocess synchronization used to construct smoothly interactive systems.

11. Dynamic

- Java is capable of linking in new class libraries, methods, and objects.
- Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at runtime. This makes it possible to dynamically link code in a safe and expedient manner.

Data Types

- The type defines set of values and set of operations can be performed on those data.
- All variables in Java must be declared before they can be used, due to this Java is termed as a *strongly typed language*.
- There are eight primitive data types in Java, as follows:
- Byte, short, int , long, float, double, char and Boolean.



Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

These can be put in four groups:

- **Integers**: This group includes **byte**, **short**, **int**, and **long**, which are for whole valued signednumbers.
- **Floating-point numbers:** This group includes **float** and **double**, which represent numbers with fractionalprecision.
- **Characters:** This group includes **char**, which represents symbols in a character set, like letters and numbers.
- **Boolean:** This group includes **boolean**, which is a special type for representing true/falsevalues.

In C and C++ allow the size of an integer to vary based upon the execution environment. Because of Java's portability requirement, all data types have a strictly defined range. For example, an **int** is always 32 bits, regardless of the particular platform

a. Integers:

Java defines four integer types: **byte**, **short**, **int**, and **long**. All of these are signed, positive and negative values. **Byte**s and **short**s are 32-bit values to improve performance, The width and ranges of these integer types vary widely, as shown in this table:

Name	**width**	**Range**
Long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
Int	32	–2,147,483,648 to 2,147,483,647
byte	8	–128 to 127
short	16	–32,768 to 32,767

Byte:

The smallest integer type is byte. This is a signed 8-bit type that has a range from –128 to 127. Variables of type byte are useful when working with a stream of data from a network or file Byte variables are declared by use of the byte keyword.

Example: byte b, c;

short

short is a signed 16-bit type. It has a range from –32,768 to 32,767. It is probably the least-used Java type, since it is defined as having its high byte first (called *big-endian* format).

Example: short s;
short t;

int

The most commonly used integer type is **int**. It is a signed 32-bit type that has a range from –2,147,483,648 to 2,147,483,647. Variables of type **int** are commonly employed to control loops and to index arrays. The **int** type is the most versatile and efficienttype.

Long:

long is a signed 64-bit type and is useful for those occasions where an **int** type is not large enough to hold the desired value. The range of a **long** is quite large.

b. Floating-PointTypes

Floating-point numbers, also known as *real* numbers, are used when evaluating expressions that require fractional precision. For example: square root of a value. There are two kinds of floating-point types, **float** and **double**, which represent single- and double-precision numbers, respectively.

Name	Width	Range
Double	64	4.9e–324 to 1.8e+308
Float	32	1.4e–045 to 3.4e+038

float

The type **float** specifies a *single-precision* value that uses 32 bits of storage. Variables of type **float** are useful when you need a fractional component, but don't require a large degree of precision.

Example: float hightemp, lowtemp;

double

Double precision, as denoted by the **double** keyword, uses 64 bits to store a value. All transcendental math functions, such as **sin( )**, **cos( )**, and **sqrt( )**, return **double** values. When you need to maintain accuracy over many iterative calculations **double** is the bestchoice.

Example: A java program to find area of a circle

```
class Area{
    public static void main(String
        args[]) { double pi, r, a;
        r = 10.8;
        pi = 3.1416;
        a = pi * r *r;
        System.out.println("Area of circle is " + a);
    }
}
```

c. character types

In Java, the data type used to store characters is **char**. In C/C++, **char** is an integer type that is 8 bits wide. Instead, Java uses Unicode to represent characters. In Java **char** is a 16-bit type. The range of a **char** is 0 to 65,536. There are no negative **char**s.

Example:

```
class CharDemo{
        public static void main(String
                args[]){ char ch1, ch2;
                ch1 = 88; //
                code for X ch2
                = 'Y';
                System.out.print("ch1 and ch2: ");
                System.out.println(ch1 + " " +ch2);
        }
}
```

This program displays the output: ch1 and ch2: XY

d. Booleans

Java has a simple type, called **boolean**, for logical values. It can have only one of two possible values, **true**or **false**. This is the type returned by all relational operators, such as **a < b**.

Example:

Boolean result= true;

```
//          Demonstrate
boolean values.  class
BoolTest {
        public static voidmain(Stringargs[])        {
        booleanb;
                b = false; System.out.println("b is " + b); b =true;
                System.out.println("b is " + b);
                // a boolean value can control the if statement if(b)
                        System.out.println("This is executed."); b = false;
                if(b)
                        System.out.println("This is not executed.");
                // outcome of a relational operator is a boolean value System.out.println("10 > 9 is "
                + (10 > 9));
        }
}
```

The output generated by this program is shown here: b is false
b is true
This is executed
.
10 > 9 is true

Variable

- The variable is the basic unit of storage in a Java program.
- A variable is defined by the combination of an identifier, a type, and an optional initializer.

Declaring a Variable :

- A variable must be declared before it is used. The basic form of a variable declaration is:
  type identifier [= *value*][, *identifier* [= *value*] ...] ;
- The type is one of the types(int, float, char, etc) or name of the class, interface, etc.,
- The *identifier* is the name of the variable.
- The variable can be initialized by specifying an equal sign and a value.
- Keep in mind that the initialization expression must result in a value of the same type as that specified for the variable

Example:

```
int a, b, c;
int d = 3, e, f = 5; // declares three more ints, initializing
byte z = 22; // initializes z.
double pi = 3.14159; // declares an approximation of pi.
char x = 'x'; // the variable x has the value 'x'.
```

Types of Variables and its Scope:

There are three types of variables.

1. Instance Variables
2. Class Variables
3. Local Variables

Now, let us dig into the scope and lifetime of each of the above mentioned type.

1.Instance Variables

A variable which is declared inside a class, but is declared outside any methods and blocks is known as *instance* variable.
Scope: Throughout the class except in the static methods.
Lifetime: Until the object of the class stays in the memory.

2.Class Variables

A variable which is declared inside a class, outside all the blocks and is declared as *static* is known as *class* variable.
Scope: Throughout the class.
Lifetime: Until the end of the program.

3.Local Variables

All variables which are not *instance* or *class* variables are known as *local* variables.
Scope: Within the block it is declared.
Lifetime: Until control leaves the block in which it is declared.

Example:

```
public class scope_and_lifetime {
   int num1, num2;   //Instance Variables
   static int result;  //Class Variable
   int add(int a, int b){  //Local Variables
      num1 = a;
      num2 = b;
      return a+b;
   }
   public static void main(String args[]){
      scope_and_lifetime ob = new scope_and_lifetime();
      result = ob.add(10, 20);
      System.out.println("Sum = " + result);
   }
}
```

Output:
Sum=30

Arrays

- An array is a collection of variables of homogeneous type that occupy contiguous memory locations.
- Arrays of any type can be created and may have one or more dimensions.
- A specific element in an array is accessed by its index.
- Arrays offer a convenient means of grouping related information. In Java all arrays are dynamically allocated.

One-Dimensional Arrays

- An array with one dimension is called one-dimensional array.
- Obtaining an array is a two-step process in java. declare a variable of the desired array type.

The general form of a one dimensional array declaration is :

```
type var-name[ ];
```

allocate the memory that will hold the array, using new, and assign it to the array variable. The general form of allocating memory using new is:

```
array-var = new type[size];
```

Here, type specifies the type of data being allocated, size specifies the number of elements in the array, and array- var is the array variable that is linked to the array. The **elements in the array allocated by new will automatically be initialized to zero.**

Example: int marks[];

```
marks = new int[90];
```

The above two steps can also be combined into one step as follows;

```
int marks[]=new int[20];
```

Accessing Array elements:

One can access a specific element in the array by specifying its index. All array indexes start at zero. For example, to assign the value 55 to the second element of marks use,

```
marks[1] = 55;
```

To display the value stored at index 3.

```
System.out.println(marks[3]);
```

Initialization

Arrays can be initialized when they are declared. An array initializer is a list of comma-separated expressions surrounded by curly braces.

```
int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Note: If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), cause a run-time error.

Example: Write a java program to find a given element using linear search

```
import java.util.*;
import java.io.*;
class linearsearch {
    public static void main(String args[])throws IOException    {
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter n value");
```

```
                int n=Integer.parseInt(br.readLine());
                int a[]=new int[n];
                System.out.println("Enter array elements");
                for(int i=0;i<n;i++)
                        a[i]=Integer.parseInt(br.readLine());
                        System.out.println("Enter search element");
                        int s=Integer.parseInt(br.readLine());
                        boolean found=false;
                        for(int i=0;i<n;i++) if(a[i]==s)          {
                                found=true; break;
                        }
                        if(found==true)
                                System.out.println("Element Found");
                        else
                                System.out.println("Element not Found");
        }
}
```

Multidimensional Arrays

An array with more than one dimension is called as a multidimensional array.An array with two dimensions is called a two-dimensional array.

For Example consider the following statement,

```
int a[][] = new int[4][5];
```

This creates an array with 4 rows and 5 columns. Internally this matrix is implemented as an array of arrays ofint. Conceptually, this array will look like the one shown below.

Example: To illustrate two dimensional arrays

```
class TwoDArray {
        public static void main(String args[]) {
                int a[][]= new int[4][5];
                int i, j, k = 0;
                for(i=0; i<4; i++)       {
                        for(j=0; j<5; j++) {
                                a[i][j] = k; k++;
                        }
                }

                for(i=0; i<4; i++) {
                        for(j=0; j<5; j++)
                                System.out.print(a[i][j] + " ");
                        System.out.println();
                }
        }
}
```

When memory is allocated for a multidimensional array, we need only specify the memory for the first(leftmost) dimension. We can allocate the remaining dimensions separately.

For example, this following code allocates memory for the first dimension of a when it is declared. It allocates the second dimension manually.

int a[][] = new int[4][]; a[0] = new int[5];
a[1] = new int[5];
a[2] = new int[5];
a[3] = new int[5];
We can manually allocate differing size second dimensions. int b[][] = new int[4][];
b[0] = new int[1];
b[1] = new int[2];
b[2] = new int[3];
b[3] = new int[4]; The array b looks like below

It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces. We can use expressions as well as literal values inside of array initializers.

```
double m[][] = { {0.0, 1.0, 2.0, 3.5},
{0.4, 1.3, 2.0, 3.4},
{0.1, 1.5, 2.5, 3.5},
{0.1, 1.1, 2.2, 3.3}
};
```

Example : Implement a Java program to multiply two given matrices.

```
import java.lang.*;
import java.io.*;
class matmul{
    public static void main(String args[])throws IOException{
        BufferedReader bf=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Enter dimensions of first matrix");
        int m=Integer.parseInt(bf.readLine());
        int n=Integer.parseInt(bf.readLine());
        System.out.println("Enter dimensions of second matrix");
        int p=Integer.parseInt(bf.readLine());
        int q=Integer.parseInt(bf.readLine());

        if(n==p)        {
            int a[][]=new int[m][n];
            int b[][]=new int[p][q];
            int c[][]=new int[m][q];

            System.out.println("Enter values of the first matrix");
```

```
                for(int i=0;i<m;i++)
                        for(int j=0;j<n;j++)
                                a[i][j]=Integer.parseInt(bf.readLine());

                System.out.println("Enter values of the second matrix");
                for(int i=0;i<p;i++)
                        for(int j=0;j<q;j++)
                                b[i][j]=Integer.parseInt(bf.readLine());

                for(int i=0;i<m;i++) {
                        for(int j=0;j<q;j++) {
                                c[i][j]=0;
                                for(int k=0;k<n;k++)
                                        c[i][j]=c[i][j]+a[i][k]*b[k][j];

                                System.out.print(c[i][j]+"\t");
                        }
                        System.out.println();
                }
        }
```

Operators

- Most of its operators can be divided into the following four groups: arithmetic, bitwise, relational, and logical. Java also defines some additional operators that handle certain special situations

Arithmetic Operators:

- Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra.
- The basic arithmetic operations—addition, subtraction, multiplication, and division
- The minus operator also has a unary form that negates its single operand

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
– –	Decrement

```
// Demonstrate the basic arithmetic operators. class BasicMath {
        public static voidmain(Stringargs[])        {
                // arithmetic using integers System.out.println("IntegerArithmetic"); int a = 1 +1;
                int b = a *3;
                int c = b / 4; int d = c - a; int e = -d;
                System.out.println("a = " + a);
                System.out.println("b = " + b);
                System.out.println("c = " + c);
                System.out.println("d = " + d);
                System.out.println("e = " + e);
        }
}
```

When you run this program, you will see the following output:

```
Integer Arithmetic a =2
b =6
c =1
d = -1
e = 1
```

Arithmetic Operators : Modulus operator

- The modulus operator (%), returns the remainder of a division operation. It can be
- applied to floating-point types as well as integer types.

```
class Modulus {
public static void main(String args[]){
        int x = 42;
```

```
double y = 42.25;
  System.out.println("x mod 10 = " + x % 10);
  System.out.println("y mod 10 = " + y % 10);
}
}
```

Output:

```
x mod 10 = 2
y mod 10 = 2.25
```

Arithmetic Operators : Compound Assignment Operators

- Java provides special operators that can be used to combine an arithmetic operation with an assignment.
- For example, if we want to perform following operation

  a=a+5;
- We can write the above operation as below also.

  a+=5;
- We can perform similar operations on other arithmetic operators also.
- The compound assignment operators provide two benefits. First, they save you a bit of typing,
- Second, they are implemented more efficiently by the Java run-time system than are their equivalent long forms.

```
class OpEquals {
  public static void main(String args[]) {
      int a = 1, b = 2, c = 3;
      a += 5;
      b *= 4;
      c += a * b;
      c %= 6;
      System.out.println("a = " + a);
      System.out.println("b = " + b);
      System.out.println("c = " + c);
  }
}
```

Output:

```
a = 6
b = 8
c = 3
```

Arithmetic Operators : Increment and Decrement

- The ++ and the – – are Java's increment and decrement operators.
- The **increment** operator increases its operand by **one.**
- The **decrement** operator decreases its operand by **one**.
- For example, the following statement x = x + 1;
- can also be rewritten like below. x++;
- Similarity, , the following statement x = x - 1;

- can also be rewritten like below. X--;
- These operators appears as postfix form or prefix form.
- In the prefix form, the operand is incremented or decremented before the value is obtained for use in the expression.
- In postfix form, the previous value is obtained for use in the expression, and then the operand is modified.

Arithmetic Operators : Increment and Decrement

- For example,

X=42;

Y= ++x;

- In the above example, y is set to **43**.

Y=x++;

- In the above example, y is set to **43** only.
- Similarly,

X=42;

Y= --x;

- In the above example, y is set to **41**.

Y=x--;

- In the above example, y is set to **41** only.
- For example,

X=42;

Y= ++x;

- In the above example, y is set to **43**.

Y=x++;

- In the above example, y is set to **42** only.
- Similarly,

X=42;

Y= --x;

- In the above example, y is set to **41**.

Y=x--;

- In the above example, y is set to **41** only.

```
class IncrDecr{
    public static void main(String[] args){
      int x=42,y;
         y=++x;
      System.out.println("X value :"+x);
      System.out.println("Y value :"+y);
      y=x++;
      System.out.println("X value :"+x);
      System.out.println("Y value :"+y);
      System.out.println("---------------------------------------");
      x=42;              y=--x;
      System.out.println("X value :"+x);
      System.out.println("Y value :"+y);
      y=x--;
      System.out.println("X value :"+x);
      System.out.println("Y value :"+y);
   }
}
```

Output:

```
X value :43
Y value :43
X value :44
Y value :43
-------------------------------------------
X value :41
Y value :41
X value :40
Y value :41
```

Relational Operators:

- The relational operators determine the relationship that one operand has to the other.
- The outcome of these operations is a boolean value.
- The relational operators are most frequently used in control statements.

Operator	Result

==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Bitwise Operators:

- Java defines several bitwise operators that can be applied to the integer types, long, int, short, byte, and char .
- These operators act upon the individual bits of their operands.

Operator	Result
~	Bitwise Unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
!=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

The Bitwise Logical Operators:

- The bitwise logical operators are &, |, ^, and ~. The following table shows the outcome of each operation.

A	B	A|B	A&B	A^B	~A
0	0	0	0	0	1

1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

The Bitwise NOT

Also called the *bitwise complement*, the unary NOT operator, **~**, inverts all of the bits of its operand. For example, the number 42, which has the following bit pattern:

00101010

becomes

11010101 after the NOT operator is applied.

The Bitwise AND

The AND operator, **&**, produces a 1 bit if both operands are also 1. A zero is produced in all other cases. Here is an example

00101010 42

&00001111 15

00001010 10

The Bitwise OR

The OR operator, |, combines bits such that if either of the bits in the operands is a 1, then the resultant bit is a 1, as shownhere:

0010101042

| 00001111 15

0010111147

The Bitwise XOR

The XOR operator, ^, combines bits such that if exactly one operand is 1, then the result is 1. Otherwise, the result is zero.

0010101042

^0000111115

0010010137

The Left Shift

The left shift operator, <<, shifts all of the bits in a value to the left a specified number of times. It has this general form:

value << *num*

Here, *num* specifies the number of positions to left-shift the value in *value*. That is, the <<moves all of the bits in the specified value to the left by the number of bit positions specified by *num*.

For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right. This means that when a left shift is applied to an **int** operand, bits are lost once they are shifted past bit position

31. If the operand is a **long**, then bits are lost after bit position 63.

Since each left shift has the effect of doubling the original value, programmers frequently use this fact as an efficient alternative to multiplying by 2.

```
// Left shifting as a quick way to multiply by2. class MultByTwo{
        public static voidmain(Stringargs[])            {
        int i;
                int num = 0xFFFFFFE; for(i=0; i<4; i++) {
                        num = num << 1; System.out.println(num);
                }
        }
}
```

The program generates the following output:

536870908
1073741816
2147483632
-32

The Right Shift

The right shift operator, >>, shifts all of the bits in a value to the right a specified number of times. Its general form is shown here:

value >> *num*

Here, *num* specifies the number of positions to right-shift the value in *value*. That is, the >>moves all of the bits in the specified value to the right the number of bit positions specified by*num.*
When a value has bits that are "shifted off," those bits are lost. For example, the next code fragment shifts the value 35 to the right two positions, which causes the two low-order bits to be lost, resulting again in **a** being set to 8.

```
int a = 35;
a = a >> 2; // a still contains 8
```

Looking at the same operation in binary shows more clearly how thishappens: 0010001135
>> 2000010008

Each time you shift a value to the right, it divides that value by two—and discards any remainder. the previous contents of the top bit. This is called *sign extension* and serves to preserve the sign of negative numbers when you shift them right. For example, –8 >> 1 is –4, which, in binary, is

11111000 –8
>> 111111100 –4

It is interesting to note that if you shift –1 right, the result always remains –1, since sign extension keeps bringing in more ones in the high-order bits.

The Unsigned Right Shift

As you have just seen, the >>operator automatically fills the high-order bit with its previous contents each time a shift occurs. This preserves the sign of the value. However, sometimes this is

undesirable.

For example, if you are shifting something that does not represent a numeric value, you may not want sign extension to take place. This situation is common when you are working with pixel-based values and graphics. In these cases, you will generally want to shift a zero into the high-order bit no matter what its initial value was. This is known as an *unsigned shift.*

To accomplish this, you will use Java's unsigned, shift-right operator, >>>, which always shifts zeros into the high-order bit.

The following code fragment demonstrates the >>>. Here, **a** is set to –1, which sets all 32 bits to 1 in binary. This value is then shifted right 24 bits, filling the top 24 bits with zeros, ignoring normal sign extension. This sets **a** to255.

```
int a = -1;
a = a >>> 24;
```

Here is the same operation in binary form to further illustrate what is happening:

```
11111111 11111111 11111111 11111111 –1 in binary as an int
        >>>24
00000000 00000000 00000000 11111111 255 in binary as an int
```

```
// Demonstrate the boolean logical operators. class BoolLogic {
    public static void main(String args[]){ boolean a =true;
        boolean b = false; boolean c = a |b;
        boolean d = a & b; boolean e = a ^ b;
        boolean f = (!a & b) | (a & !b);
        boolean g = !a;
        System.out.println(" a = " + a);
        System.out.println(" b = " + b);
        System.out.println(" a|b = " + c);
        System.out.println(" a&b = " + d);
        System.out.println(" a^b = " + e);
        System.out.println("!a&b|a&!b = " + f);
        System.out.println(" !a = " + g);
    }
}
```

Output:

```
a = true
b = false
a|b = true
a&b = false
a^b =true
a&b|a&!b = true
!a = false
```

Short-Circuit Logical Operators

Java provides two interesting Boolean operators not found in many other computer languages. These are secondary versions of the Boolean AND and OR operators, and are known as *short-circuit* logical operators.

If you use the || and **&&**forms, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone. This is very useful when the right-hand operand depends on the value of the left one in order to function properly.

For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:

```
if (denom != 0 && num / denom > 10)
```

Since the short-circuit form of AND (**&&**) is used, there is no risk of causing a run-time exception when **denom** is zero. If this line of code were written using the single **&**version of AND, both sides would be evaluated, causing a run-time exception when **denom** is zero.

The Assignment Operator

The assignment operator is used to assign a value to variable.The *assignment operator* is the single equal sign, **=**.
It has this general form:

var = expression;

Here, the type of *var* must be compatible with the type of *expression.*

it allows you to create a chain of assignments. For example

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

Using a “chain of assignment” is an easy way to set a group of variables to a common value.

The ? Operator

Java includes a special *ternary* (three-way) *operator* that can replace certain types of if-then-else statements. This operator is the **?**.

The **?** has this general form:

expression1 **?** *expression2* **:** *expression3*

Here, *expression1* can be any expression that evaluates to a **boolean** value. If *expression1* is **true**, then
expression2 is evaluated; otherwise, *expression3* is evaluated.

```
class Ternary{
```

```
        public static voidmain(Stringargs[])        { int i, k;
                i = 10;
                k = i < 0 ? -i : i; // get absolute
                value          of          i
                System.out.print("Absolute
                value of "); System.out.println(i
                + " is " + k);
        }
}
```

Output:

Absolute value of 10 is 10

Java has well-defined rules for specifying the order in which the operators in an expression are evaluated when the expression has several operators. For example, multiplication and division have a higher precedence than addition and subtraction. Precedence rules can be overridden by explicit parentheses.

First Simple Program Entering the Program

Java is a case sensitive language. The name you give to a source file is very important. For this example, the name of the source file should be **Example.java**. The Java compiler requires that a source file use the **.java** filename extension.

```
/*  simple  java
program        */
import
java.lang.*;
class Example{
        public static void main (String args[]){
                System.out.println("Welcome to java");
        }
 }
```

The name of the class defined by the program is also **Example**. In Java, all code must reside inside a class. By convention, the name of that class should match the name of the file that holds the program.

Comments

The contents of a comment are ignored by the compiler. Java supports three styles of comments.

- Single line comment://
- Multilinecomment:/* .....*/
- Documentationcomment:/** ***/.** This type of comment is used to producean HTML file that documents your program. This type of comment is readable to both, computer and human.

Defining a class

This line uses the keyword **class** to declare that a new class is being defined. **Example** is an *identifier* that is the name of the class. The entire class definition, including all of its members, will be between the opening curly brace and the closing curly brace ({, }).

public static void main(String args[])

This is the line at which the program will begin executing. All Java applications begin execution by calling **main( )**.

The **public** keyword is an *access specifier*. When a class member is preceded by **public**, then that member may be accessed by code outside the class in which it is declared. (private and protected are other access specifiers).

The function **main( )** must be declared as **public**, since it must be called by code outside of its class when the program is started.

The keyword **static** allows **main( )** to be called without having to instantiate a particular instance of the class. This is necessary since **main( )** is called by the Java interpreter before any objects are made.

Java compiler will compile classes that do not contain a **main( )** method. But the Java interpreter has no way to run these classes. In applets you won't use **main( ).**

The keyword **void** simply tells the compiler that **main( )** does not return a value.

In **main( )**, there is only one parameter. **String args[ ]** declares a parameter named **args**, which is an array of instances of the class **String**. (*Arrays* are collections of similar objects.) Objects of type **String** store character strings. In this case, **args** receives any command-line arguments present when the program is executed.

To display output

System.out.println("Welcome to Java");

This line outputs the string "Welcolme to java" followed by a new line on the screen. Output is actually accomplished by the built-in **println( )** method. In this case, **println( )** displays the string which is passed to it. The line begins with **System.out**. in which **System** is a predefined class that provides access to the system, and **out** is the output stream that is connected to theconsole.

Note:

- Every statement in Java ends with asemicolon.
- Java is a case sensitivelanguage.

System.out.println()

num = 100;

System.out.println("This is num: " + num);

The first line declares an integer variable called **num** assigns to **num** the value 100. The next line of code outputs the value of **num** preceded by the string "This is num:".

In this statement, the plus sign causes the value of **num** to be appended to the string that precedes it, and then the resulting string is output. (Actually, **num** is first converted from an integer into its string equivalent and then concatenated with the string that precedesit.

Using the + operator, you can string together as many items as you want within a single **println( )** statement.
The built-in method **print( )** is just like **println( )**, except that it does not output a newline character System.out.print("Example to display output ");

Note

Both **print( )** and **println( )** can be used to output values of any of Java's built-in types.

Compiling and executing the Program

To compile the **Example** program, execute the following command at command prompt:

C:\>javac Example.java

The **javac** compiler creates a file called **Example.class** that contains the bytecode version of the program. The Java bytecode is the intermediate representation of a program that contains instructions the Java interpreter will execute. The output of **javac** is not code that can be directly executed.
To actually run the program, use the Java interpreter, called **java**. To do so, pass the class name **Example**
as a command-line argument, as shown here:

C:\>java Example

When the program is run, the following output is displayed:
Welcome to java

Expressions

- An expression is a sequence of **operands** and **operators** that reduces to a single value.
- An expression can be **simple** or **complex**.
- An **operator** takes an action.
- An **operand** is an object on which an operation is performed. Every language has operators whose actions are clearly specified in the language syntax.
- A **Simple expression** contains only **one operator**. For example, **2+5**. and **-a**.
- A complex expression contains more than one operator. For example, 2+5*7.

Control Statements

Control statements can be put into the following categories: **selection, iteration, and jump**.

- ***Selection*** statements allow your program to choose different paths of execution based upon the outcome of an expression or the state of avariable.
- ***Iteration*** statements enable program execution to repeat one or more statements (that is, iteration statements formloops).
- ***Jump*** statements allow your program to execute in a nonlinearfashion.

Java's Selection Statements

Java supports two selection statements: **if** and **switch**. These statements allow you to control the flow of your program's execution based upon conditions known only during run time.

If:

The **if** statement is Java's conditional branch statement. It can be used to route program execution through two different paths.

Here is the general form of the **if** statement: if (*condition*)
statement1;
else *statement2*;

Here, each *statement* may be a single statement or a compound statement enclosed in curly braces (that is, a
block). The *condition* is any expression that returns a **boolean** value. The **else** clause is optional.

The **if** works like this: If the *condition* is true, then *statement1* is executed. Otherwise, *statement2* (if it exists) is executed. In no case will both statements be executed. For example, consider the following:

```
int a, b;
// ...
if(a < b) a = 0; else b = 0;
```

Here, if **a** is less than **b**, then **a** is set to zero. Otherwise, **b** is set to zero. In no case are they both set to zero. Most often, the expression used to control the **if** will involve the relational operators.
However, this is not technically necessary. It is possible to control the **if** using a single **boolean** variable.

```
class BoolIf {
        public static voidmain(Stringargs[])        {
        booleanb;
                b = false;
                System.out.println("b is " + b);
                b =true;
                System.out.println("b is " + b);
                // a boolean value can control the if statement if(b)
```

```
                System.out.println("This is executed."); b = false;
        if(b)
                System.out.println("This is not executed.");
        // outcome of a relational operator is a boolean value System.out.println("10 > 9 is " +
        (10 > 9));
    }
}
```

The output generated by this program is shown here:

```
b is false b is true
This is executed.
10 > 9 is true
```

Nested ifs

A*nested* **if** is an **if** statement that is the target of another **if** or **else**. Nested **if**s are very common in programming. When you nest **if**s, the main thing to remember is that an **else** statement always refers to the nearest **if** statement that is within the same block as the **else** and that is not already associated with an **else**. Here is an example:

```
if(i = = 10){
    if(j < 20) a = b;
    if(k > 100) c = d; // this if is
    else a = c; // associated with this else
}
else a = d; // this else refers to if(i == 10)
```

As the comments indicate, the final **else** is not associated with **if(j<20)** because it is not in the same block (even though it is the nearest **if** without an **else**). Rather, the final **else**is associated with **if(i==10)**. The inner **else** refers to **if(k>100)** because it is the closest **if** within the sameblock.

The if-else-if Ladder

A common programming construct that is based upon a sequence of nested **if**s is the *if-else-if ladder*. It looks like this:

```
    if(condition)
            statement;
   elseif(condition)
            statement;
  elseif(condition)
            statement;
            ...
    else
            statement;
```

The **if** statements are executed from the top down. As soon as one of the conditions controlling the **if** is**true**, the statement associated with that **if** is executed, and the rest of the ladder is bypassed.

If none of the conditions is true, then the final **else** statement will be executed. The final **else** acts as a default condition; that is, if all other conditional tests fail, then the last **else** statement is performed. If there is no final **else** and all other conditions are **false**, then no action will take place.

```
class Test{
        public static voidmain(Stringargs[])            {
        int x =30;
                if( x ==0 )                    {
                    System.out.print("Value of X is 0");
                }
                else if( x>0)                  {
                    System.out.print("Value of X is Positive");
                }
                else if( x <0 )                {
                    System.out.print("Value    of    X
                    isNegative");
                }
        }
  }
//  Demonstrate  if-else-if
statements. class IfElse {
     public static voidmain(Stringargs[])  {
        int month=4;       // April Stringseason;
        if(month == 12 || month == 1 || month == 2) season = "Winter";
      else if(month == 3 || month == 4 || month == 5) season = "Spring";
      else if(month == 6 || month == 7 || month == 8) season = "Summer";
      else if(month == 9 || month == 10 || month == 11) season = "Autumn";
      else
            season = "Bogus Month";
      System.out.println("April is in the " + season + ".");
   }
}
```

Here is the output produced by the program:

April is in the Spring.

switch

The **switch** statement is Java's multiway branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements.

Here is the general form of a **switch** statement:

switch (*expression*){

```
        case value1:
                    // statement sequence
                break;

        case value2:
                // statement sequence
                break;
            …

        case valueN:
                // statement sequence
                break;
        default:
            // default statement sequence

}
```

The *expression* must be of type **byte**, **short**, **int**, or **char**; each of the *values* specified in the **case** statements must be of a type compatible with the expression.

Each **case** value must be a unique literal (that is, it must be a constant, not a variable). Duplicate **case** values are not allowed.

The **switch** statement works like this:

The value of the expression is compared with each of the literal values in the **case** statements. If a match is found, the code sequence following that **case** statement is executed. If none of the constants matches the value of the expression, then the **default** statement is executed. However, the **default** statement is optional. If no **case** matches and no **default** is present, then no further action istaken.

The **break** statement is used inside the **switch** to terminate a statement sequence. When a **break** statement is encountered, execution branches to the first line of code that follows the entire **switch** statement. This has the effect of "jumping out" of the**switch**.

```
// A simple example of the
switch. class SampleSwitch{
        public static voidmain(Stringargs[])            { for(int i=0; i<6;i++)
                switch(i){
                    case 0: System.out.println("i is zero.");
                            break;
                    case1:
                                System.out.println("i is one.");
                                break;
                    case2:
                                System.out.println("i is two.");
```

```
                    break;
            case3:
                    System.out.println("i is three.");
                    break;
            default:
                    System.out.println("i is greater than3.");
        }
    }
}
```

The output produced by this program is shown here:

```
i is zero.
i is one.
i is two.
i is three.
i is greater han 3.
i is greater than 3.
```

Iteration Statements

Java's iteration statements are **for**, **while**, and **do-while**. These statements create *loops. A* loop repeatedly executes the same set of instructions until a termination condition is met.

while

The **while** loop is Java's most fundamental loop statement. It repeats a statement or block while its controlling expression is true.

Here is its general form:

```
while(condition)        {
    // body of loop
}
```

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 passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

```
class WhileDemo {
    public static voidmain(Stringargs[])        { int n =5;
        while(n>0)          {
            System.out.println("Value = " + n);
            n--;
        }
    }
}
```

Output:

```
Value=1
```

Value=2
Value=3
Value=4
Value=5

do-while

Since the **while** loop evaluates its conditional expression at the top of the loop, the body of the loop will not execute even once if the condition is initially false (the body of the loop will not be executed atall).

However, sometimes it is desirable to execute the body of a loop at least once, even if the conditional expression is false initially. In other words, there are times when you would like to test the termination expression at the end of the loop rather than at the beginning. To serve this purpose java supplies ***do-while*** loop.

The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop.

Its general form is

```
do              {
        // body of loop
} while (condition);
```

Each iteration of the **do-while** loop first executes the body of the loop and then evaluates the conditional expression. If this expression is true, the loop will repeat. Otherwise, the loop terminates.

```
class DoWhile {
    public static voidmain(Stringargs[])        {
    int n =5;
        do {
            System.out.println("Value = " + n);
            n--;
        } while(n > 0);
    }
}
```

The **do-while** loop is especially useful when you process a menu selection, because you will usually want the body of a menu loop to execute at least once.

```
import java.util.scanner;
public class ConsoleMenuDemo {
  public static void main(String[] args) {
    // Local variable
```

```
    int choice,a,b;
    Scanner s=new Scanner(System.in);
do{
        System.out.println("   MENU SELECTION DEMO ");
        System.out.println("============================");
        System.out.println(" 1.addition "); System.out.println(" 2.subtraction ");
        System.out.println(" 3.multiplication");
        System.out.println("============================");
        choice = s.nextInt();
        }while(choice<1 || choice >3);
        switch (choice)          {
                case 1:
                        System.out.println(" enter value of a and b");
                        a = s.nextInt();
                        b = s.nextInt();
                        System.out.println("a+b ="+(a+b));
                        break;

                case 2:
                        System.out.println(" enter value of a and b");
                        a = s.nextInt();
                        b = s.nextInt();
                        System.out.println("a-b ="+(a-b));
                        break;

                case 3:
                        System.out.println(" enter value of a and b");
                        a = s.nextInt();
                        b = s.nextInt(); System.out.println("a*b ="+(a*b));
                        break;
                default:
                        System.out.println("Invalid selection");
                        break; // This break is not really necessary

        }
}
}
```

for:

There are two forms of the **for** loop. The first is the traditional form that has been in use since the original version of Java. The second is the new "for-each" form.

the general form of the traditional **for** statement:

```
for(initialization;condition;iteration)      {
        // body
```

```
}
```

If only one statement is being repeated, there is no need for the curly braces.

The **for** loop operates as follows.

When the loop first starts, the *initialization* portion of the loop is executed. Generally, this is an expression that sets the value of the *loop control variable*, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once.

Next, *condition* is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.

Next, the *iteration* portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable.

The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

```
class ForTick {
        public static voidmain(Stringargs[])            { int n;
                for(n=5; n>0; n--)
                        System.out.println("tick = " + n);
        }
}
```

Output:

tick =5
tick =4
tick =3
tick =2
tick =1

Declaring Loop Control Variables Inside the for Loop

Sometimes, the variable that controls a **for** loop is only needed for the purposes of the loop and is not used elsewhere. When this is the case, it is possible to declare the variable inside the initialization portion of the **for**.

```
// Declare a loop control variable inside
the for. class ForTick {
        public static voidmain(Stringargs[])            {
                // here, n is declared inside of the for
                loop for(int n=5; n>0; n--)
                        System.out.println("tick = " + n);
        }
```

}

Output: tick =5
tick =4
tick =3
tick =2
tick =1

When you declare a variable inside a **for** loop, there is one important point to remember:

The scope of that variable ends when the **for** statement does. (That is, the scope of the variable is limited to the **for** loop.) Outside the **for** loop, the variable will cease to exist. If you need to use the loop control variable elsewhere in your program, you will not be able to declare it inside the **for** loop.

The **For-Each** Version of the for Loop**:**

A foreach style loop is designed to cycle through a collection of objects, such as an array, in strictly sequential fashion, from start to finish. The for-each style of **for** is also referred to as the *enhanced* **for** loop.

The general form of the for-each version of the **for** is shown here:

for(*type itr-var : collection*) *statement-block*

Here, *type* specifies the type and *itr-var* specifies the name of an *iteration variable* that will receive the elements from a collection, one at a time, from beginning to end. There are various types of collections that can be used with the **for.**

Example:

```
// Use a for-each style
for loop. class ForEach
        public static voidmain(Stringargs[])            {
        int nums[] = { 1, 2, 3, 4, 5};
                int sum = 0;
                // use for-each style for to display and sum thevalues
                for(int x:nums)       {
                        System.out.println("Value is: " + x);
                        sum += x;
                }
                System.out.println("Summation: " + sum);
        }
}
```

The output from the program is shown here. Value is:1
Value is:2
Value is:3
Value is:4
Value is:5
Summation: 55

As this output shows, the for-each style for **automatically cycles through an array in sequence from the lowest index to the highest**.

With each pass through the loop, **x** is automatically given a value equal to the next element in **nums**. Thus, on the first iteration, **x** contains 1; on the second iteration, **x** contains 2; and so on.
Not only is the syntax streamlined, but it also **prevents boundary errors.**

Although the for-each **for** loop iterates until all elements in an array have been examined, it is possible to terminate the loop early by using a **break** statement.

For example, this program sums only the first five elements of **nums**:

```
// Use break with a for-each
style for. class ForEach2 {
        public static voidmain(Stringargs[])        {
                int sum =0;
                int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
                // use for to display and sum the values
                for(int x:nums){
                        System.out.println("Value is: " + x);
                        sum += x;
                        if(x == 5) break; // stop the loop when 5 is obtained
                }
                System.out.println("Summation of first 5 elements: " + sum);
        }
}
```

This is the output produced:

```
Value is:1
Value is:2
Value is:3
Value is:4
Value is:5
Summation of first 5 elements: 15
```

There is one important point to understand about the for-each style loop.

Its iteration variable is “read-only” as it relates to the underlying array. An assignment to the iteration variable has no effect on the underlying array. In other words, you can’t change the contents of the array by assigning the iteration variable a new value.

Example:

```
// The for-each loop is essentially
read-only. class NoChange {
        public static voidmain(Stringargs[])        {
                int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
                for(int x:nums)                {
```

```
                System.out.print(x + "");
                x = x * 10; // no effect on nums
            }
            System.out.println(); for(int x : nums)
                System.out.print(x + " "); System.out.println();
        }
}
```

The first **for** loop increases the value of the iteration variable by a factor of 10. However, this assignment has no effect on the underlying array **nums**, as the second **for** loop illustrates.

The output, shown here, proves this point: 1 2 3 4 5 6 7 8 9 10

1 2 3 4 5 6 7 8 9 10

Iterating Over Multidimensional Arrays

The enhanced version of the **for** also works on multidimensional arrays. Remember, multidimensional arrays consist of *arrays of arrays*. (For example, a two-dimensional array is an array of one-dimensional arrays.)

This is important when iterating over a multidimensional array, because each iteration obtains the *next array*, not an individual element. Furthermore, the iteration variable in the **for** loop must be compatible with the type of array beingobtained.

For example, in the case of a two-dimensional array, the iteration variable must be a reference to a one- dimensional array. In general, when using the for-each **for** to iterate over an array of *N* dimensions, the objects obtained will be arrays of *N*–1 dimensions. To understand the implications of this, consider the following program. It uses nested **for** loops to obtain the elements of a two-dimensional array in roworder, from first tolast.

```
// Use for-each style for on a two-dimensionalarray. class ForEach3{
    public static void main(String args[]) { int sum = 0,k=0;
        int nums[][] = new int[3][5];
        // give nums some values for(int i = 0; i < 2; i++)
            for(int j=0; j < 3; j++)
                nums[i][j] = k++;
                // use for-each for to display and sum the values
            for(int x[]: nums)      {
                for(int y: x)    {
                System.out.println("Value is: " + y);
                sum += y;
            }
```

```
                }
                System.out.println("Summation: " + sum);
}
}
```

The output from this program is shown here:

```
Value is:0
Value is:1
Value is:2
Value is:3
Value is:4
Value is:5
Summation: 15
```

In the program, pay special attention tothisline: **for(int x[] :nums)**

Notice how **x** is declared. It is a reference to a one-dimensional array of integers. This is necessary because each iteration of the **for** obtains the next *array* in **nums**, beginning with the array specified by **nums[0]**. The inner **for** loop then cycles through each of these arrays, displaying the values of eachelement.

```
// Search an array using for-each
style for. class Search {
        public static voidmain(Stringargs[])          {
                int nums[] = { 6, 8, 3, 7, 5, 6, 1, 4};
                int val = 5;
                boolean found = false;
                // use for-each style for to search nums forval
                 for(int x:nums)      {
                        if(x==val){
                                found = true;
                                break;
                        }
                }
                if(found)
                        System.out.println("Value found!");
        }
}
```

Jump Statements

Java supports three jump statements: **break**, **continue**, and **return**. These statements transfer control to another part of your program.

break

In Java, the **break** statement has three uses.

1. It terminates a statement sequence in a **switch**statement.
2. It can be used to exit aloop.

3. **break** can be used as a "civilized" form of goto. The last two uses are explainedhere.

Using break to Exit a Loop:

- By using **break**, you can force immediate termination of a loop, bypassing the conditional expression and any remaining code in the body of theloop.

- When a **break** statement is encountered inside a loop, the loop is terminated, and program control resumes at the next statement following theloop.

```
// Using break to
exit a loop. class
BreakLoop {
        public static voidmain(Stringargs[])            {
                for(int i=0; i<100;i++){
                        if(i == 5) break; // terminate loop if i is 10
                        System.out.println("i = " + i);
                }
                System.out.println("Loop complete.");
        }
}
```

This program generates the following output:

```
i = 0
i =1
i = 2
i = 3
i = 4
Loop complete.
```

Continue:

Sometimes it is useful to force an early iteration of a loop. That is, you might want to continue running the loop but stop processing the remainder of the code in its body for this particular iteration. The **continue** statement performs such an action.

In **while** and **do-while** loops, a **continue** statement causes control to be transferred directly to the conditional expression that controls the loop.

```
class ContinueEx{
        public static void main(String[] args){
        int i,n=10;
        for(i=1;i<=n;i++){
                if(i==5)
                        continue;
        System.out.println("i value is: "+i);
        }
    }
}
```

Output:

```
i value is: 1
i value is: 2
i value is: 3
i value is: 4
i value is: 6
i value is: 7
i value is: 8
i value is: 9
i value is: 10
```

Return:

The **return** statement is used to explicitly return from a method. That is, it causes program control to transfer back to the caller of the method.

At any time in a method the **return** statement can be used to cause execution to branch back to the caller of the method. Thus, the **return** statement immediately terminates the method in which it is executed.

The following example illustrates this point. Here, **return** causes execution to return to the Java run-time system, since it is the run-time system that calls **main()**.

```
// Demonstrate return.
class Return {
        public static voidmain(Stringargs[])            {
                boolean t =true;

                System.out.println("Before the return.");
                if(t)
                        return; // return tocaller
                System.out.println("This won't execute.");
        }
}
```

Output:

Before the return.

As you can see, the final **println( )** statement is not executed. As soon as **return** is executed, control passes back to the caller.

One last point: In the preceding program, the **if(t)** statement is necessary. Without it, the Java compiler would flag an "unreachable code" error because the compiler would know that the last **println( )** statement would never be executed. To prevent this error, the **if** statement is used here to trick the compiler for the sake of this demonstration.

Type Conversion andCasting

Java's AutomaticConversions

When one type of data is assigned to another type of variable, an *automatic type conversion* will take place if the following two conditions are met:

1. The two types arecompatible.
2. The destination type is larger than the sourcetype.

When these two conditions are met, a *widening conversion* takes place. For example, the **int** type is always large enough to hold all valid **byte** values, so no explicit cast statement is required.

For widening conversions, the numeric types, including integer and floating-point types, are compatible with each other.

However, there are no automatic conversions from the numeric types to **char** or **boolean**. Also, **char** and **boolean** are not compatible with each other.

As mentioned earlier, Java also performs an automatic type conversion when storing a literal integer constant into variables of type **byte**, **short**, **long**, or **char**.

Casting Incompatible Types

Although the automatic type conversions are helpful, they will not fulfill all needs. For example, what if you want to assign an **int** value to a **byte** variable? This conversion will not be performed automatically, because a **byte** is smaller than an **int**. This kind of conversion is sometimes called a *narrowing conversion,* since you are explicitly making the value narrower so that it will fit into the target type.

To create a conversion between two incompatible types, you must use a cast. A *cast* is simply an explicit type conversion.

It has thisgeneralform: **(*target-type*)*value***

Here, *target-type* specifies the desired type to convert the specified value to. For example, the following fragment casts an **int** to a **byte**. If the integer's value is larger than the range of a **byte**, it will be reduced modulo (the remainder of an integer division by the) **byte**'s range.

```
int a; byte b;
// ...
b = (byte) a;
```

A different type of conversion will occur when a floating-point value is assigned to an integer type: *truncation*. As you know, integers do not have fractional components. Thus, when a floating-point value is assigned to an integer type, the fractional component is lost.

For example, if the value 1.23 is assigned to an integer, the resulting value will simply be 1.
The 0.23 will have been truncated. Of course, if the size of the whole number component is too large to fit into the target integer type, then that value will be reduced modulo the target type's range.

The following program demonstrates some type conversions that require casts:

```
// Demonstrate casts.
class Conversion {
        public static voidmain(Stringargs[])            {
                byteb;
                int i = 257;
                double d = 323.142;

                System.out.println("\nConversion of int to byte.");
                b = (byte) i;
                System.out.println("i and b " + i + " " + b);

                System.out.println("\nConversion of double to int.");
                i = (int) d;
                System.out.println("d and i " + d + " " + i);


                System.out.println("\nConversion of double to byte."); b = (byte) d;
                System.out.println("d and b " + d + " " + b);
        }
}
```

Output:

```
Conversion of int to byte. i and b 257 1
Conversion of double to int. d and i 323.142 323
Conversion of double to byte. d and b 323.14267
```

Let's look at each conversion. When the value 257 is cast into a **byte** variable, the result is the remainder of the division of 257 by 256 (the range of a **byte**), which is 1 in this case.

When the **d** is converted to an **int**, its fractional component is lost. When **d** is converted to a **byte**, its fractional component is lost, *and* the value is reduced modulo 256, which in this case is 67.

Simple Java Program

The requirement for Java Hello World Example

For executing any Java program, the following software or application must be properly installed.

- Install the JDK if you don't have installed it, download the JDK and install it.
- Set path of the jdk/bin directory. http://www.javatpoint.com/how-to-set-path-in-java
- Create the Java program
- Compile and run the Java program

Example

```
class Simple{
    public static void main(String args[]){
     System.out.println("Hello Java");
    }
}
```

Compile by: javac Simple.java

Run by: java Simple

Output:

Hello Java

Module-2

Syllabus: Basics of Java

Classes and Objects - Concepts of Classes, Objects, Constructors, Methods, This Keyword, Garbage Collection, Overloading Methods and Constructors, Parameter Passing, Recursion, String Handling: String, String Buffer, String Tokenizer.

Inheritance - Base Class Object, Subclass, Member Access Rules, Super keyword, final keyword, Method Overriding, Abstract Classes.

Class fundamentals

The class is at the core of Java. The class is a logical construct which defines the shape and nature of an object.

A class defines a new data type. Once defined, this new type can be used to create objects of that type. Thus, a class is a template for an object, and an object is an instance of a class.

The General Form of a Class

A class contains both the data and the code that operates on that data. A class is declared by use of the class keyword. The general form of a class definition is:

```
class classname {
        type instance-variable1; type instance-variable2;
        // ...
        type instance-variableN;
        type methodname1(parameter-list)    {
                // body of method
        }
        type methodname2(parameter-list)    {
                // body of method
        }
        // ...
        type methodnameN(parameter-list)    {
                // body of method
        }
}
```

The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables. Thus, the data for one object is separate and unique from the data for another.

A Simple Class Example:

Here is a class called Box that defines three instance variables: width, height, and depth. Currently, Box does not contain any methods.

```
class Box {
        double width, height, depth;
}
```

A class defines a new type of data. In this case, the new data type is called Box. This name can be used to declare objects of type Box. It is important to remember that a class declaration only creates a template; it does not create an actual object.

To actually create a Box object, you will use a statement like the following:

```
Box mybox = new Box(); // create a Box object called mybox
```

After this statement executes, mybox will be an instance of Box. Thus, it will have "physical" reality. Every Box object will contain its own copies of the instance variables width, height, and depth.

To access variables, use the dot (.) operator along with the object name. The dot operator links the name of the object with the name of an instance variable. To assign a value to the data member the syntax is as follows.

Object . datamember;

Example: to assign the width variable of mybox the value 100, you would use the following statement: mybox.width = 100;

```
/* A program that uses the Box class. Call this file BoxDemo.java
*/
class Box {
        double width; double height; double depth;
}
// This class declares an object of type Box.
class BoxDemo {
        public static void main(String args[]) {
                Box mybox = new Box(); double vol;
                // assign values to mybox's instance variables
                mybox.width = 10;
                mybox.height = 20;
                mybox.depth = 15;
                // compute volume of box
                vol = mybox.width * mybox.height * mybox.depth;
                System.out.println("Volume is " + vol);
        }
}
```

We should save this file that contains this program BoxDemo.java, because the main( ) method is in the class called BoxDemo, not in the class called Box.

To run this program, you must execute BoxDemo.class. When you do, you will see the following Output:

Volume is 3000.0

Example2: To illustrate creation of class and object

```
class student {
        int no;
        char grade; float percent;
}
class demo {
        public static void main(String args[]) {
                student s=new student();
                s.no=10; s.grade='c';
                s.percent=98;
                System.out.println("number ="+s.no);
                System.out.println("grade ="+s.grade);
                System.out.println("percent ="+s.percent);
        }
}
```

Output:

```
Number =10
Grade=c
percent=98
```

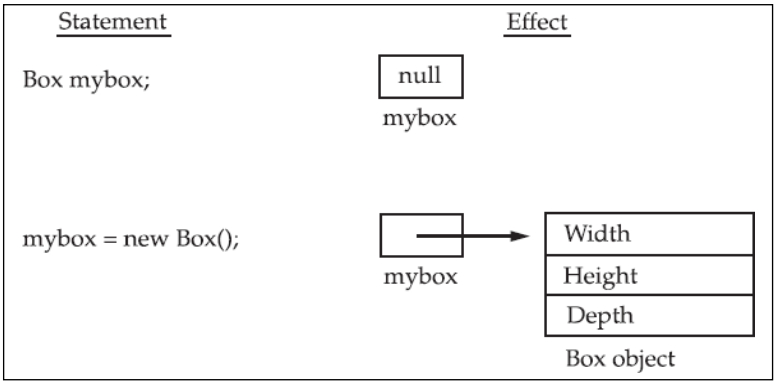
We should save this file that contains this program demo.java, because the main( ) method is in the class called demo, not the class called student.

Declaring Objects:

With a class definition a new data type is created. We can use this type to declare objects of that type. Obtaining objects of a class is a two-step process.

- First, declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
- Second, acquire an actual, physical copy of the object and assign it to that variable. This can be done using the new operator. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it.

This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. This reference is then stored in the variable.



Example:

```
Box mybox; // declare reference to object
mybox = new Box(); // allocate a Box object
```

The first line declares mybox as a reference to an object of type Box. After this line executes, mybox contains the value null, which indicates that it does not yet point to an actual object. Any attempt to use mybox at this point will result in a compile-time error.

The next line allocates an actual object and assigns a reference to it to mybox. After the second line executes, you can use mybox as if it were a Box object. But in reality, mybox simply holds the memory address of the actual Box object.

The effect of these two lines of code is depicted in the figure as follows

The above two statements can also be combines into a single statement as follows

```
Box mybox = new Box();
```

The new operator:

The new operator dynamically allocates memory for an object. It has the general form:

```
class-var = new classname( );
```

Here, class-var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines what occurs when an object of a class is created.

It is important to understand that new allocates memory for an object during run time. The advantage of this approach (using new) is that the program can create as many or as few objects as it needs during the execution of the program.

The distinction between a class and an 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.)

Assigning Object Reference Variables:

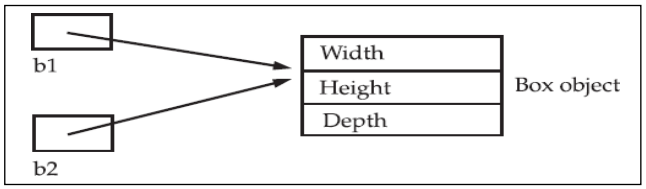
Object reference variables act differently than you might expect when an assignment takes place. For example:

```
Box b1 = new Box();
Box b2 = b1;
```

Here b2 is being assigned a reference to a copy of the object referred to by b1. That means b1 and b2 will both refer to the same object.

The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object. This situation is depicted here:

For example, a subsequent assignment to b1 will simply unhook b1 from the original object without affecting the object or affecting b2.



For example:

```
Box b1 = new Box();
Box b2 = b1;
// ...
b1 = null;
```

Here, b1 has been set to null, but b2 still points to the original object.

Introducing Methods

Classes usually consist of two things: instance variables and methods. In most classes, the instance variables are acted upon and accessed by the methods defined for that class. Thus the methods that determine how a class' data can be used.

The general form of a method is:

```
type name(parameter-list) {
// body of method
}
```

- Here, type specifies the type of data returned by the method. If the method does not return a value, its return type must be void.
- The name of the method is specified by name.
- The parameter-list is a sequence of type and identifier pairs separated by commas.

Methods that have a return type other than void return a value to the calling routine using the following form of the return statement:

return value;

Here, value is the value returned.

Adding a Method to the Box Class:

Methods provide access to data. In addition to defining methods that provide access to data, we can also define methods that are used internally by the class itself.

// This program includes a method inside the box class. class Box {	**Output:** Volume is 3000.0

```
        double width, height, depth;
// display volume of a box
        void volume(){
                System.out.print("Volume is ");
                System.out.println(width * height * depth);
        }
}
class BoxDemo {
        public static void main(String args[]) {
                Box b1 = new Box();
                // assign values to mybox1's instance variables
                b1.width = 10;
                b1.height = 20;
                b1.depth = 15;
                // display volume of box
                b1.volume();
        }
}
```

b1.volume(); invokes the volume( ) method ,that is, it calls volume( ) relative to the b1 object, using the object's name followed by the dot operator.

When b1.volume( ) is executed, the Java run-time system transfers control to the code defined inside volume( ). After all the statements inside volume ( ) have executed, control is returned to the calling routine, and execution resumes with the line of code following the call.

Returning a Value:

A better way to implement volume( ) is to have it compute the volume of the box and return the result to the caller.

```
class Box {
        double width, height, depth;
// compute and return volume double
        volume() {
                return (width * height * depth);
        }
}
class BoxDemo
 {
        public static void main(String args[]) {
                Box b1 = new Box();
                Double vol;
                // assign values to mybox1's instance variables b1.width =
                10;
                b1.height = 20;
                b1.depth = 15;
                // display volume of box vol=b1.volume();
                System.out.println("Volume is " + vol);
        }
}
```

Output:

Volume is 3000.0

There are two important things to understand about returning values:

- The type of data returned by a method must be compatible with the return type specified by the method. For example, if the return type of some method is boolean, you could not return an integer.
- The variable receiving the value returned by a method (such as vol, in this case) must also be compatible with the return type specified for the method.

Adding a Method That Takes Parameters:

Parameters allow a method to be generalized. That is, a parameterized method can operate on a variety of data and/or be used in a number of slightly different situations.

```
class Box {
        double width, height, depth;
        // compute and return volume
        double volume()        {
                return (width * height * depth);
        }
        // sets dimensions of box
        void setDim(double w, double h, double d) {
                width = w;      height = h;      depth = d;
        }
}
class BoxDemo {
        public static void main(String args[]) {
                Box b1 = new Box();
                Double vol;
                b1.setDim(10, 20, 15);          // initialize box
                System.out.println("Volume is " + b1.volume());
        // display volume of box
        }
}
```

Output:

```
Volume is 3000.0
```

- The setDim( ) method is used to set the dimensions of each box.
  For example, when mybox1.setDim(10, 20, 15); is executed, 10 is copied into parameter w, 20 is copied into h, and 15 is copied into d.
- Inside setDim( ) the values of w, h, and d are then assigned to width, height, and depth, respectively.

It is important to keep the two terms parameter and argument straight.

A parameter is a variable defined by a method that receives a value when the method is called.

For example, setDim(double w, double h, double d) → where w, h, d are parameters

An argument is a value that is passed to a method when it is invoked.

For example, setDim(10, 20, 15);passes 10,20,30 as arguments.

Constructors:

It can be tedious to initialize all of the variables in a class each time an instance is created.

Automatic initialization of data members of a class is performed through the use of a constructor.

- A constructor initializes an object immediately upon creation.
- A Constructor has the same name as the class name.
- Constructor syntactically similar to a method.

- Constructor is automatically called when the object is created before the new operator completes.
- Constructors have no return type, not even void. This is because the implicit return type of a class' constructor is the class type itself.

```
/* Here, Box uses a constructor to initialize the dimensions of a box. */
class Box {
        double width, height, depth;
        // This is the constructor for Box.
        Box() {
                System.out.println("Constructing Box");
                width = 10;     height = 10;     depth = 10;
        }
        // compute and return volume
        double volume()         {
                return width * height * depth;
        }
}
class BoxDemo {
        public static void main(String args[]) {
                // declare, allocate, and initialize Box objects
                Box b1 = new Box(); Box b2 = new Box();
                double vol;
                vol = b1.volume();      // get volume of first box
                System.out.println("Volume is " + vol);
                vol = b2.volume();      // get volume of second box
                System.out.println("Volume is " + vol);
        }
}
```

Output:

```
Constructing Box
Constructing Box
Volume is 1000.0
Volume is 1000.0
```

let's re-examine the new operator. As you know, when you allocate an object, you use the following general form:

class-var = new classname( );

Now you can understand why the parentheses are needed after the class name. What is actually happening is that the constructor for the class is being called. Thus, in the line

Box b1 = new Box();

new Box( ) is calling the Box( ) constructor. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class.

The default constructor automatically initializes all instance variables to zero. The default constructor is often sufficient for simple classes, but it usually won't do for more sophisticated ones. Once you define your own constructor, the default constructor is no longer used.

Parameterized Constructors:

The default constructor (Box()) in the preceding example does initialize a Box object. But it is not very useful because all boxes have the same dimensions. What is needed is a way to construct Box objects of various dimensions. The easy solution is to add parameters to the constructor.

A constructor which takes parameters is called a parameterized constructor

```
class Box {
        double width, height, depth;
// This is the constructor for Box.
     Box(double w, double h, double d)
        {
                width = w;      height = h;      depth = d;
        }
        // compute and return volume
        double volume()
        {
                return width * height * depth;
        }
}
class BoxDemo {
        public static void main(String args[])
        {
                // declare, allocate, and initialize Box objects
                 Box b1 = new Box(10, 20, 15);
                Box b2 = new Box(3, 6, 9); double vol;
                // get volume of first box vol = b1.volume();
                System.out.println("Volume is " + vol);
                // get volume of second box vol = b2.volume();
                System.out.println("Volume is " + vol);
        }
}
```

Output:

```
Volume is 3000.0
Volume is 162.0
```

Each object is initialized as specified in the parameters to its constructor.
For example, in the following line, Box b1 = new Box(10, 20, 15);
the values 10, 20, and 15 are passed to the Box( ) constructor when new creates the object. Thus, b1's copy of width, height, and depth will contain the values 10, 20, and 15, respectively.

this Keyword:

The this Keyword and Instance Variable Hiding

- The keyword 'this' is used inside any method to refer to the current object.
- That is, 'this' keyword is always a reference to the object on which the method was invoked.
- You can use this anywhere a reference to an object of the current class' type is permitted.
- Whenever the formal parameters and data members of a class are similar, to differentiate data members of a class from formal parameters, the data members of the class are preceded by "this".

```
class Test      {
        int a,b;
        Test(int a,int b) { //parameterized constructor
                System.out.println(" In parameterized constructor");
                this.a=a;
                this.b=b;
                System.out.println("a= "+a+" this.a= "+this.a);
                this.a=this.a+10;
                System.out.println("a= "+a+" this.a= "+this.a);
        }
        void display() {
                System.out.println(a);// Internally treated as this.a
        System.out.println(this.a);
        }
}
class ThisDemo {
        public static void main(String a[])     {
                Test t=new Test(100,200); t.display();
        }
}
```

Output:

```
a= 100   this.a= 100
a= 100   this.a= 110
110
110
```

Garbage Collection

- Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.

- In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator.

- Java takes a different approach which handles deallocation automatically. The technique that accomplishes this is called garbage collection.

- Garbage collection works like this: when no references to an object exist, that object is

assumed to be no longer needed, and the memory occupied by the object can be reclaimed.

- Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

- Different Java run-time implementations will take varying approaches to garbage collection, but for the most part, you should not have to think about it while writing your programs.

The finalize( ) Method

Sometimes an object will need to perform some action when it is destroyed. To do that Java provides a mechanism called finalization. To add a finalizer to a class, simply define the finalize() method. The Java run time calls that method whenever it is about to recycle an object of that class. The finalize( ) method has this general form:

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

Here, the keyword protected is a specifier that prevents access to finalize( ) by code defined outside its class. It is important to understand that finalize( ) is only called just prior to garbage collection. It is not called when an object goes out-of-scope, for example. This means that you cannot know when—or even if—finalize( ) will be executed.

Overloading Methods:

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. This process is referred to as method overloading.
- Method overloading is one of the ways that Java implements polymorphism.
- When an overloaded method is invoked, Java uses the type and/or number of arguments to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters.
- While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Here is a simple example that illustrates method overloading: Example: To illustrate the concept of method overloading in java

```
class Overload {
        void test()     {
                System.out.println("TEST with out parameters");
        }
        void test(int a) {
                System.out.println("parameters:"+a);
        }
        void test(int a,char ch){
                System.out.println("parameters:"+a+"\t"+ch);
        }
        int test(char ch)       {
                System.out.println("TEST with return type and parameters :"+ch);
                return 100;
        }
}
class OverloadDemo {
        public static void main(String args[]) {
                Overload o=new Overload();
                o.test();
                o.test(10); o.test(10,'x');
                int x=o.test('a'); System.out.println("X value is:"+x);
        }
}
```

In the above example test( ) is overloaded four times.

- The first version takes no parameters, the second takes one integer parameter, the third takes two parameters; one integer and one character, and the fourth takes one character parameter and also returns an integer.
- When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters.
- In some cases, Java's automatic type conversions can play a role. Java will employ its automatic type conversions only if no exact match is found.

Overloading Constructors:

Constructor methods can also be overloaded in the same as how methods are overloaded.

/* Here, Box defines three constructors to initialize the dimensions of a box various ways.

```
class Box {
        double width, height , depth;
// constructor used when all dimensions specified
        Box(double w, double h, double d)    {
                width = w;      height = h;      depth = d;
        }
```

Output:

```
Volume of mybox1 is
3000.0
Volume of mybox2 is
-1.0
Volume of mycube is
```

// constructor used when no dimensions specified
Box() {
width = -1; // use -1 to indicate
height = -1; // an uninitialized
depth = -1; // box
}
// constructor used when cube is created
Box(double len) {
width = height = depth = len;
}
// compute and return volume
double volume() {
return width * height * depth;
}
}
class OverloadCons {
public static void main(String args[]) {
// create boxes using the various constructors
Box mybox1 = new Box(10, 20, 15);
Box mybox2 = new Box();
Box mycube = new Box(7); double vol;
// get volume of first box
vol = mybox1.volume();
System.out.println("Volume of mybox1 is " + vol);
// get volume of second box vol = mybox2.volume();
System.out.println("Volume of mybox2 is " + vol);
// get volume of cube vol = mycube.volume();
System.out.println("Volume of mycube is " + vol);
}
}
	343.0

Using Objects as Parameters:

Normally we pass simple types as parameters to methods. We can also pass objects to methods. For example, consider the following program:

```
// Objects may be passed to methods.
class Test {
        int a, b;
        Test(int i, int j)          {
                a = i;    b = j;
        }
        // return true if o is equal to the invoking object
        boolean equals(Test o){
                if(o.a == a && o.b == b)
                        return true;
                else
                        return false;
```

```
        }
}
class PassOb {
        public static void main(String args[]) {
                Test ob1 = new Test(100, 22);
                Test ob2 = new Test(100, 22);
                Test ob3 = new Test(-1, -1);
                System.out.println("ob1 == ob2: " + ob1.equals(ob2));
                System.out.println("ob1 == ob3: " + ob1.equals(ob3));
        }
}
```

This program generates the following output:
ob1 == ob2: true
ob1 == ob3: false

- The equals( ) method inside Test compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed.
- If they contain the same values, then the method returns true. Otherwise, it returns false.
- Notice that the parameter o in equals( ) specifies Test as its type. Although Test is a class type created by the program, it is used in just the same way as Java's built-in types.

Inheritance

Inheritance is an important feature of object oriented programming. Inheritance is the property of acquiring the properties of other class. A class that is inherited is called a *superclass.* The class that does the inheriting is called a *subclass. A subclass* inherits members of superclass and adds its own, unique elements.

The general form of a **class** declaration that inherits a superclass is shown here:

```
class subclass-name extends superclass-name{
        // body of class
}
```

In java extends is the keyword used to inherit the other class. The class which inherits the properties is called sub class and the class from which the properties are inherited is called the super class.

Types of inheritance

- Single inheritance
- Multi-level inheritance
- Multiple inheritance
- Hybrid inheritance
- Heirarchy Inheritance

Single inheritance:

When a class inherits another class, it is known as a single inheritance. The following program creates a superclass called **A** and a subclass called **B**.

```
class A
 {
int i,j;
A()
{
int i=10,j=20;
}
void showij()
 {
System.out.println("i and j: " + i + " " + j);
}
}
class B extends A
{
int k;
B()
{
int i=100, j=200,k=300;
}
void showall()
{
System.out.println("i: " + i);
System.out.println("j: " + j);
System.out.println("k: " + k);
}
}
class Sample
 {
public static void main(String args[])
 {
A o1= new A();
B o2 = new B();
System.out.println("Contents of superOb: ");
o1.showij();
System.out.println("Contents of subOb: ");
o2.showall();
}
}
```

The subclass **B** includes all of the members of its superclass, **A**. This is why o2 can access **i** and **j** and can call showij( )**.**

Member Access and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as **private**. In a class hierarchy, private members remain private to their class.

```
class A  {
        private int i;
        int j;
        A() {
                i=10,j=20;
        }
        void showij()  {
                System.out.println("i and j: " + i + " " + j);
        }
}
Class B extends A{
      int k;
      B() {
        j=200,k=300;
      }
      void showjk()  {
                System.out.println("j: " + j);
                System.out.println("k: " + k);
        }
}
class Sample  {
        public static void main(String args[]) {
                A o1= new A();
                B o2 = new B();
                System.out.println("Contents of superOb: ");
                o1.showij();
                System.out.println("Contents of subOb: ");
                o2.showjk();
        }
}
```

In the above program the data member j is inherited but i is not inherited as it is a private member. A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses.

super

super has two general forms.

- is used to call the superclass' constructor.
- is used to access a member of the superclass that has been hidden by a member of a subclass.

a. Using super to Call Superclass Constructors

A subclass can call a constructor method defined by its immediate superclass by use of the following form of

super:

super(*parameter-list*);

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

Example

```
class A {
        int x,y;
        A(int p, int q) {
                x=p;
                y=q;
        }
        void showA() {
                System.out.println("x:"+x+"Y:"+y);
        }
}
```

```
class B extends A {
        int m,n;
        B(int p, int q,int r, int s) {
                super(r,s);
                M=p;
                N=q;
        }
        void showB() {
                System.out.println("m:"+m+"n:"+n);
        }
}
class supex {
        public static void main(String args[]) {
                A a=new A(10,20);
                System.out.println("USING SUPER CLASS");
                a.showA();
                B b=new B(100,200,300,400);
                System.out.println("USING SUB CLASS"); b.showA();
                b.showB();
        }
}
```

In the above example super keyword is used to call the constructor in the super class. Parameters can also be passed to the constructor by enclosing them with in parentheses.

A Second Use for super:

Super can also be used to refer to the superclass of the subclass in which it is used. This usage has the following general form:

super.*member*

Here, *member* can be either a method or a data member. This second form of **super** is most applicable to situations in which member names of a subclass hide members by the same name in the superclass.

Example:

```
Class A { int x;
        A() {
                x=20;
        }
        void showA() {
                System.out.println("x:"+x);
        }
}
class B extends A {
        int x;
        B()  {
                super.x=100; x=200;
        }
        void showB() {
                System.out.println("super.x:"+super.x+"x:"+x);
```

```
        }
}
class supex2 {
        public static void main(String args[]) {
                B b=new B();
                b.showB();
        }
}
```

Although the instance variable **x** in **B** hides the **x** in **A**, **super** allows access to the **x** defined in the superclass. As you will see, **super** can also be used to call methods that are hidden by a subclass.

Creating a Multilevel inheritance

In multi-level inheritance a subclass is used as a super class of another class. For example, given three classes called **A**, **B**, and **C**, **C** can be a subclass of **B**, which is a subclass of **A**. When this type of situation occurs, each subclass inherits all of the traits found in all of its super classes.

Example:

```
class A {
        int x;
        A() {
                x=20;
        }
        void showA() {
                System.out.println("x:"+x);
        }
}
class B extends A {
        int y;
        B() {
                y=30;
        }
        void showB() {
                System.out.println("y:"+y);
        }
}
class C extends B {
        int z;
        C() {
                z=40;
        }
        void showC() {
                System.out.println("z:"+z);
        }
}
class multi{
        public static void main(String args[]) {
```

```
        C c=new C();
        c.showA();
        c.showB();
        c.showC();
}
}
```

The subclass C inherits properties in the classes A and B. So it can access the methods showA(), showB() and showC().

Order Constructor execution with inheritance

Constructors are called in order of derivation, from superclass to subclass. Further, since super( ) must be the first statement executed in a subclass' constructor, this order is the same whether or not super( ) is used. If super() is not used, then the default or parameterless constructor of each superclass will be executed.

Example

```
class A {
    A() {
        System.out.println("Inside A's constructor.");
    }
}
class B extends A {
    B()     {
        System.out.println("Inside B's constructor.");
    }
}
class C extends B {
    C() {
        System.out.println("Inside C's constructor.");
    }
}
class multicons {
    public static void main(String args[]) {
        C c = new C();
    }
}
```

The output from this program is:

Inside A's constructor Inside B's constructor Inside C's constructor Method Overriding When a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass. 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 superclass will be hidden.

Example:

```
class A {
    int i, j;
    A(int a, int b) {
        i = a;
```

```
                j = b;
        }
        void show()     {
                System.out.println("i and j: " + i + " " + j);
        }
}
class B extends A {
        int k;
        B(int a, int b, int c)      {
                super(a, b);
                k = c;
        }
        void show()     {
                Super.show();
                System.out.println("k: " + k);
        }
}
class override {
        public static void main(String args[]) {
                B ob = new B(1, 2, 3);
                ob.show();
        }
}
```

With the call Ob.show( ), the method in the subclass B is invoked. That is, the version of show( ) inside B overrides the version declared in A. To access the superclass version of an overridden function, use super. Note: overridden methods allow Java to support run-time polymorphism.

Run time polymorphism using method overriding

Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. This is how Java implements run-time polymorphism.

Note: a superclass reference variable can refer to a subclass object.

Java uses this fact to resolve calls to overridden methods at run time. When an overridden method is called through a superclass 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 versions of an overridden method will be called.

Example:

```
class A {
        int x;
        A() {
                x=20;
        }
        void show() {
                System.out.println("x:"+x);
```

```
        }
}
class B extends A {
        int y;
        B() {
                y=30;
        }
        void show() {
                System.out.println("y:"+y);
        }
}
class C extends B {
        int z;
        C() {
                z=40;
        }
        void show() {
                System.out.println("z:"+z);
        }
}
class multi {
        public static void main(String args[]) { A o1=new A();
                B o2=new B(); C o3=new C(); A o; o=o1;
                o.show();//calls show in the class A o=o2;
                o.show();//calls show in the class B o=o3;
                o.show();//calls show in the class C
        }
}
```

This program creates one superclass called A and two subclasses of it, called B and C. Subclasses B and C override show( ) declared in A. Inside the main( ) method, objects of type A, B, and C are declared. Also, a reference of type A, called o, is declared. The program then assigns a reference to each type of object to o and uses that reference to invoke show( ).

Abstract Classes:

An abstract method is a method that is declared without an implementation (without braces and followed by a semicolon). To declare an abstract method, use this general form:

abstract type name(parameter-list);

Example :

abstract void draw();

The abstract method must be overridden by the subclass. If a class includes abstract methods, the class itself must be declared abstract. Any class that contains one or more abstract methods must also be declared abstract.

An abstract class is a class that is declared abstract. It may or may not include abstract methods. Objects can not be created for Abstract classes. Abstract classes can be inherited by other classes. To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.

```
abstract class classname {
```

```
}
```

Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be itself declared abstract.

Example:

```
abstract class A {
        int x;
        A() {
                x=20;
        }
        abstract void show();
}
class B extends A {
        int y;
        B() {
                y=30;
        }
        void show() {
                System.out.println("x:"+x);
                System.out.println("y:"+y);
        }
 }
class AbsEx {
        public static void main(String args[]) {
                B o=new B();
                o.show();
        }
}
```

It is not possible to create objects of the class A as it is an abstract class. The method show() in the class A is declared as abstract and is overridden by the class B.

final with Inheritance:

The keyword final has three uses.

- It can be used to create a named constant.
- Can be used to disallow a method from being overridden
- Can be used to prevent a class from being inherited

a. Using final to create constant data members

A variable can be declared as final. If a variable is declared as final its contents cannot be modified. A final variable must be initialized when it is declared.

For example:

final int FILE_NEW = 1; final int FILE_OPEN = 2; final int FILE_SAVE = 3; final int FILE_SAVEAS = 4; final int FILE_QUIT = 5;

In subsequent parts of the program we can use FILE_OPEN, etc., as if they were Constants. Variables declared as final do not occupy memory on a per-instance basis. Thus, a final variable is essentially a constant. The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables.

b. Using final to Prevent Overriding

To disallow a method from being overridden, use final as a modifier at the start of its declaration. Methods declared as final cannot be overridden.

Example:

```
class A  {
        final void meth() {
                System.out.println("This is a final method.");
        }
}
```

In the class A meth( ) is declared as final, it cannot be overridden in subclasses.

c. Using final to Prevent Inheritance

A class can also be prevented from being inherited. Use final to do so. Declaring a class as final implicitly declares all of its methods as final, too.

Note: It is illegal to declare a class as both abstract and final

Example

```
final class A {
        // ...
}
```

String Handling

In Java, string is basically an object that represents sequence of char values. An array of characters works same as Java string. For example:

1. **char**[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=**new** String(ch);
   is same as:
   String s="javatpoint";

Java String class provides a lot of methods to perform operations on strings such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

Generally, String is a sequence of characters. But in Java, string is an object that represents a sequence of characters. The java.lang.String class is used to create a string object.

There are two ways to create String object:

1. By string literal
2. By new keyword

1) String Literal

Java String literal is created by using double quotes. For Example:

String s="welcome";

Each time we create a string literal, the JVM checks the "string constant pool" first. If the string already exists in the pool, a reference to the pooled instance is returned. If the string doesn't exist in the pool, a new string instance is created and placed in the pool. For example:

1. String s1="Welcome";

2. String s2="Welcome";//It doesn't create a new instance

2) By new keyword

String s=**new** String("Welcome");//creates two objects and one reference variable

In such case, JVM will create a new string object in normal (non-pool) heap memory, and the literal "Welcome" will be placed in the string constant pool. The variable s will refer to the object in a heap (non-pool).

String Example.java

```
public class StringExample
{
public static void main(String args[])
{
String s1="java"; //creating string by Java string literal
char ch[]={'s','t','r','i','n','g','s'};
String s2=new String(ch);//converting char array to string
String s3=new String("example");//creating Java string by new keyword
System.out.println(s1);
System.out.println(s2);
System.out.println(s3);
}
}
```

Output:

```
Java
Strings
Examples
```

In Java, String objects are immutable. Immutable simply means unmodifiable or unchangeable. Once String object is created its data or state can't be changed but a new String object is created.

```
class Testimmutablestring{
public static void main(String args[]){
  String s="Sachin";
  s.concat(" Tendulkar");//concat() method appends the string at the end
  System.out.println(s);//will print Sachin because strings are immutable objects
}
}
```

Output: Sachin

```
class Testimmutablestring1{
public static void main(String args[]){
  String s="Sachin";
  s=s.concat(" Tendulkar");
  System.out.println(s);
}
}
```

Output: Sachin Tendulkar

There are three ways to compare String in Java:

1. By Using equals() Method
2. By Using == Operator
3. By compareTo() Method

1) **By Using equals() Method**

The String class equals() method compares the original content of the string. It compares values of string for equality. String class provides the following two methods:

- public boolean equals(Object another) compares this string to the specified object.
- public boolean equalsIgnoreCase(String another) compares this string to another string, ignoring case.

Teststringcomparison1

```
class Teststringcomparison1{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3=new String("Sachin");
   String s4="Saurav";
   System.out.println(s1.equals(s2));//true
   System.out.println(s1.equals(s3));//true
   System.out.println(s1.equals(s4));//false
 }
}
```

Output:

```
true
true
false
```

Teststringcomparison2

```
class Teststringcomparison2{
public static void main(String args[]){
   String s1="Sachin";
   String s2="SACHIN";
   System.out.println(s1.equals(s2));//false
   System.out.println(s1.equalsIgnoreCase(s2));//true
 }
}
```

Output:

```
false
true
```

2) By Using == operator

The == operator compares references not values.

```
class Teststringcomparison3{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3=new String("Sachin");
   System.out.println(s1==s2);//true (because both refer to same instance)
   System.out.println(s1==s3);//false(because s3 refers to instance created in nonpool)
  }
  }
```

Output:

```
true
false
```

3) By Using compareTo() method

The String class compareTo() method compares values lexicographically and returns an integer value that describes if first string is less than, equal to or greater than second string.

Suppose s1 and s2 are two String objects. If:

- **s1 == s2** : The method returns 0.
- **s1 > s2** : The method returns a positive value.
- **s1 < s2** : The method returns a negative value.

Teststringcomparison

```
 class Teststringcomparison4{
 public static void main(String args[]){
   String s1="Sachin";
   String s2="Sachin";
   String s3="Ratan";
   System.out.println(s1.compareTo(s2));//0
   System.out.println(s1.compareTo(s3));//1(because s1>s3)
   System.out.println(s3.compareTo(s1));//-1(because s3 < s1 )
 }
}
```

Output:

```
0
1
-1
```

String Concatenation

```
class TestStringConcatenation1
{
 public static void main(String args[])
{
   String s="Sachin"+" Tendulkar";
   System.out.println(s);//Sachin Tendulkar
 }
}
```

Output:
Sachin Tendulkar

Substring

```
public class TestSubstring
{
 public static void main(String args[])
{
 String s="SachinTendulkar";
 System.out.println("Original String: " + s);
 System.out.println("Substring starting from index 6: " +s.substring(6));//Tendulkar
 System.out.println("Substring starting from index 0 to 6: "+s.substring(0,6)); //Sachin
 }
}
```

Output:
Original String: SachinTendulkar
Substring starting from index 6: Tendulkar
Substring starting from index 0 to 6: Sachin

Java String length() Method

The String class length() method returns length of the specified String.

```
public class Stringoperation5
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.length());//6
}
}
```

Output:
6

Java String charAt() Method

The String class charAt() method returns a character at specified index.

```
public class Stringoperation4
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.charAt(0));//S
System.out.println(s.charAt(3));//h
}
}
```

Output:

```
S
h
```

Java String replace() Method

The String class replace() method replaces all occurrence of first sequence of character with second sequence of character.

```
public class Stringoperation8
{
public static void main(String args[])
{
String s1="Java is a programming language. Java is a platform. Java is an Island.";
String replaceString=s1.replace("Java","Kava");//replaces all occurrences of "Java" to "Kava"
System.out.println(replaceString);
}
}
```

Output:

```
Kava is a programming language. Kava is a platform. Kava is an Island
```

Java StringBuffer Class:

Java StringBuffer class is used to create mutable (modifiable) String objects. The StringBuffer class in Java is the same as String class except it is mutable i.e. it can be changed.

1) StringBuffer Class append() Method

The append() method concatenates the given argument with this String.

```
class StringBufferExample{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
```

```
sb.append("Java");//now original string is changed
System.out.println(sb);//prints Hello Java
}
}
```
Output:
Hello Java

2) StringBuffer insert() Method

The insert() method inserts the given String with this string at the given position.

```
class StringBufferExample2{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello ");
sb.insert(1,"Java");//now original string is changed
System.out.println(sb);//prints HJavaello
}
}
```
Output:
Hjavaello

3) StringBuffer delete() Method

The delete() method of the StringBuffer class deletes the String from the specified beginIndex to endIndex.

```
class StringBufferExample4{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.delete(1,3);
System.out.println(sb);//prints Hlo
}
}
```
Output:
Hlo

4) StringBuffer reverse() Method

The reverse() method of the StringBuilder class reverses the current String.

```
class StringBufferExample5{
public static void main(String args[]){
StringBuffer sb=new StringBuffer("Hello");
sb.reverse();
System.out.println(sb);//prints olleH
}
}
```
Output:
olleh

5) StringBuffer capacity() Method

The capacity() method of the StringBuffer class returns the current capacity of the buffer. The default capacity of the buffer is 16. If the number of character increases from its current capacity, it increases the capacity by (oldcapacity*2)+2. For example if your current capacity is 16, it will be (16*2)+2=34.

```
class StringBufferExample6{
public static void main(String args[]){
StringBuffer sb=new StringBuffer();
System.out.println(sb.capacity());//default 16
sb.append("Hello");
System.out.println(sb.capacity());//now 16
sb.append("java is my favourite language");
System.out.println(sb.capacity());//now (16*2)+2=34 i.e (oldcapacity*2)+2
}
}
```

Output:

String vs StringBuffer

java provides the string buffer and string classes, and the string class is used to manipulate character strings that cannot be changed. Simply stated objects of type string are read only and immutable. The string buffer class is used to represent characters that can be modified.

Example

```
class str {
        public static void main(String[] args) {
                String str1=new String("welocme");
                String str2=new String("java");
                String str3=str1+str2;
                System.out.println("STRING1:"+str3);
                System.out.println("Length of the string is:"+str3.length());
                System.out.println("s1==s2:"+str1.equals(str2));
                System.out.println("chartAt(3):"+str3.charAt(3));
        }
}
```

StringBuilder:

Introduced by JDK 5, StringBuilder is a recent addition to Java's string handling capabilities. StringBuilder is identical to StringBuffer except for one important difference: it is not synchronized, which means that it is not thread safe.

The advantage of StringBuilder is faster performance. However, in cases in which a mutable string will be accessed by multiple threads, and no external synchronization is employed, you must use StringBuffer rather than StringBuilder.

It has the following constructors.

Constructor	Description
StringBuilder()	creates an empty string Builder with the initial capacity of 16.
StringBuilder(String str)	creates a string Builder with the specified string.
StringBuilder(int length)	creates an empty string Builder with the specified capacity as length.

It has the following methods.

Method	**Description**
Append	is used to append the specified string with this string. The append() method is overloaded like append(char), append(boolean), append(int), append(float), append(double) etc.
Insert	is used to insert the specified string with this string at the specified position. The insert() method is overloaded like insert(int, char), insert(int, boolean), insert(int, int), insert(int, float), insert(int, double) etc.
Delete	is used to delete the string from specified startIndex and endIndex.
Reverse	is used to reverse the string.
Capacity	is used to return the current capacity.
Length	is used to return the length of the string i.e. total number of characters.
substring	is used to return the substring from the specified beginIndex, beginIndex to endIndex.

StringBuffer	**StringBuilder**
StringBuffer is synchronized i.e. thread safe. It means two threads can't call the methods of StringBuffer simultaneously.	StringBuilder is non-synchronized i.e. not thread safe. It means two threads can call the methods of StringBuilder simultaneously.
StringBuffer is less efficient than StringBuilder.	StringBuilder is more efficient than StringBuffer.

StringTokenizer in Java:

The **java.util.StringTokenizer** class allows you to break a String into tokens. It is simple way to break a String. It is a legacy class of Java.

StringTokenizer Class

```
import java.util.StringTokenizer;
public class Simple
{
public static void main(String args[])
{
StringTokenizer st = new StringTokenizer("my name is khan"," ");
while (st.hasMoreTokens())
{
        System.out.println(st.nextToken());
    }
  }
```

```
}
```
Output:
```
my
name
is
khan
```

nextToken(String delim) method
```
import java.util.*;
public class Test {
  public static void main(String[] args) {
     StringTokenizer st = new StringTokenizer("my,name,is,khan");

    // printing next token
    System.out.println("Next token is : " + st.nextToken(","));
  }
}
```
Output:
```
my
```

hasMoreTokens() method
```
import java.util.StringTokenizer;
public class StringTokenizer1
{
 /* Driver Code */
 public static void main(String args[])
 {
   /* StringTokenizer object */
   StringTokenizer st = new StringTokenizer("Demonstrating methods from StringTokenizer class",
" ");
    /* Checks if the String has any more tokens */
    while (st.hasMoreTokens())
    {
       System.out.println(st.nextToken());
    }
 }
}
```
Output:
```
Demonstrating
methods
from
StringTokenizer
class
```

Module-3

Syllabus:

Interfaces - Defining an interface, implementing interface, differences between classes and interfaces, extending interfaces. Packages - Defining, creating, and accessing a package, access control, exploring package-java.io (file handling).

Exception handling - Concepts of Exception handling, benefits of exception handling, exception hierarchy, checked and unchecked exceptions, usage of try, catch, throw, throws and finally, built-in exceptions, creating own exception subclasses

Interfaces:

In general, an interface specifies what operations must permit users to perform but does not specify how the operations are performed. For example, remote serves as an interface between television and user.

Software objects also communicate via interfaces. An interface is syntactically similar to the class. An interface declaration begins with the keyword interface and contains constant data members and method declarations. An interface may not specify any implementation details. All methods declared in an interface are implicitly public abstract methods and all data members are implicitly public, static and final.

Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```
Access-specifier interface name {
        return-type method-name1(parameter-list);
        return-type method-name2(parameter-list);
       //..
        return-type method-nameN(parameter-list);
        type final-varname1 = value;
        type final-varname2 = value;
        // ...
        type final-varnameN = value;
}
```

Here, access is either public or none. name is the name of the interface. The methods which are declared in the interface have no bodies. Variables declared inside of interface are implicitly final and static. They must be initialized with a constant value.

Here is an example of an interface definition.

```
interface inter {
        int a=10;
        void show();
}
```

Use of interfaces

Interfaces can not be instantiated. They can be inherited to other interfaces or to other classes. Using interfaces the concept of multiple inheritances can be achieved. Once an interface is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces. Each class that includes an interface must implement all of the methods declared in the interface. The data members declared in the interface cannot be changed by the implementing class.

Implementing Interfaces

Once an 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 defined by the interface. The general form of a class that includes the implements clause looks like this:

```
access class classname [extends superclass] [implements interface [,interface...]] {
// class-body
}
```

Here, access is either public or none. If a class implements more than one interface, the interfaces are separated with a comma. The methods that implement an interface must be

declared public. Also, the type signature of the implementing method must match exactly the type signature specified in the interface definition.
Here is a small example class that implements the example interface shown earlier.

```
class sample implements inter {
        public void disp() {
                System.out.println("Example interface");
                System.out.println(a);
        }
}
```

classes that implement interfaces can define additional members of their own. For example:

```
class sample implements example {
        public void disp() {
                System.out.println("Example interface");
                System.out.println(a);
        }
        void show()    {
                System.out.println(“Example method in class”);
        }
}
```

Accessing Implementations through Interface References

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. The following example calls the disp() method via an interface reference variable:

```
class demo {
        public static void main(String args[]) {
                inter i1 = new sample();
                i1.disp(42);
        }
}
```

Notice that variable i1 is declared to be of the interface type inter, yet it was assigned an instance of the class sample. Although i1 can be used to access the disp() method, it cannot access any other members of the sample class. An interface reference variable only has knowledge of the methods declared by its interface declaration. Thus, i1 could not be used to access show( ) since it is defined by sample Partial Implementations

If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract. For example:

```
abstract class A implements inter {
        int a, b;
        void show() {
                System.out.println(a + " " + b);
        }
        // ...
}
```

Here, the class A does not implement disp() and must be declared as abstract. Any class that inherits A must implement disp( ) or be declared abstract itself.

Interfaces Can Be Extended

interface can inherit another interface by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

```
interface A {
        void meth1();
        void meth2();
}
interface B extends A {
        void meth3();
}
class Sample implements B {
        public void meth1() {
                System.out.println("Implement meth1().");
        }
        public void meth2() {
                System.out.println("Implement meth2().");
        }
        public void meth3() {
                System.out.println("Implement meth3().");
        }
}
class demo {
        public static void main(String arg[]) {
                sample ob = new sample();
                ob.meth1();
                ob.meth2();
                ob.meth3();
        }
}
```

A is an interface which contains the declarations of meth1() and meth2(). B is an interface which extends A and contains the declaration of meth3(). In the class Sample, which implements interface B all the methods meth1(), meth2() and meth3() must be defined. Any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

Packages:

A Java source-code file must have the following order:

1. a package declaration (if any),
2. import declarations (if any), then
3. class declarations.

Only one of the class declarations in a particular file can be public.

Packages

In java a package is a collection of classes. Groups of related classes are often packaged as reusable components. Java packages are of two types:

- Predefined packages
- User defined packages

Benefits of packages:

- As applications become more complex, package helps to manage the complexity of application components.
- Packages facilitate software reuse by enabling programs to import classes from other packages.
- Packages provide a convention for unique class names, which helps prevent class-name conflicts.

Predefined packages

Java provides a rich set of predefined classes that are grouped into categories of related classes called packages. These packages are referred as the Java Application Programming Interface (Java API), or the Java class library. These packages can be imported into programs and reused.

Some packages in java API

java.lang: The Java Language Package contains classes and interfaces that are required by many Java programs. This package is imported by the compiler into all programs, need not to use import keyword.

java.io: The Java Input/Output Package contains classes and interfaces that enable programs to input and output data.

java.util: The Java Utilities Package contains utility classes and interfaces that enable such actions as date and time manipulations, random number processing (class Random), the storing and processing of large amounts of data and the breaking of strings into smaller pieces called tokens (class StringTokenizer).

java.applet: The Java Applet Package contains a class and several interfaces required to create Java applets—programs that execute in Web browsers.

java.awt: The Java Abstract Window Toolkit Package contains the classes and interfaces required to create and manipulate GUIs.

Defining a Package (user defined)

Include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. If the package statement is omitted, the class names are put into the default package, which has no name. This is the general form of the package statement:

package pkg;

Here, pkg is the name of the package. For example, the following statement creates a package called pack1. package pack1;

- Java uses file system directories to store packages. The .class files for any classes that are

declared as part of pack1 must be stored in a directory called pack1.

- More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong.
- We can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.
- The general form of a multileveled package statement is shown here: package pkg1[.pkg2[.pkg3]];

Finding Packages and CLASSPATH

For example, consider the following package specification. package pack1;

In order for a program to find pack1, one of two things must be true. Either the program is executed from a directory immediately above pack1, or CLASSPATH must be set to include the path to pack1.

A Short Package Example

```
package pack1;
class A {
        int  x;  int  y;
        A() {
                x=10; y=100;
        }
        void fun()      {
                System.out.println("x:"+x+"y:"+y);
        }
}
class packex {
        public static void main(String args[]) {
                A o=new A();
                o.fun();
        }
}
```

Save this file as packex.java, in a directory called pack1. Next, compile the file. Make sure that the resulting .class file is also in the pack1 directory. Then try executing the packex.java class, using the following command line:

>java pack1.packex.java

Be in the directory above pack1 to execute this command. It cannot be executed by itself as it is a part of the package pack1. That is packex must be qualified with its package name.

Access Protection:

Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package
- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

The three access specifiers, private, public, and protected, provide a variety of ways to

produce the many levels of access required by these categories. Anything declared public can be accessed from anywhere. Anything declared private cannot be seen outside of its class. When a member does not have an explicit access specification (default), it is visible to subclasses as well as to other classes in the same package. A protected member can be accessed outside the current package, but only to the sub classes.

Access Specifier	private	No modifier	Protected	Public
Same class	√	√	√	√
Same package sub class		√	√	√
Same package non-subclass		√	√	√
Different package subclass			√	√
Different package non-subclass				√

A class has only two possible access levels: default and public. When a class is declared as public, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. Importing Packages

All of the standard classes in java are stored in some named package. Java includes the import statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name. In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions. The general form of the import statement:

import pkg1[.pkg2].(classname|*);

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.). Finally, specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package.

Example:

```
import java.util.Date;
import java.io.*;
```

All of the standard Java classes included with Java are stored in a package called java. Java.lang package is implicitly imported by the compiler for all programs. When a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code.

Example: A program to import user defined package

A.java

```
package pack1;
public class A {
        int x,y;
        public A(){
                x=10; y=100;
        }
        public void dispA(){
                System.out.println("In class A");
```

```
                System.out.println("x:"+x+"y:"+y);
        }
}
```

B.java

```
package pack1;
public class B {
        int m,n;
        public B(){
                m=100; n=200;
        }
        public void dispB()     {
                System.out.println("In class B");
                System.out.println("m:"+m+"n:"+n);
        }
}
```

demopack.java

```
import pack1.*;
class demopack {
        public static void main(String args[]) {
                A o1=new A();
                B o2=new B(); O1.dispA();
                o2.dispB();
        }
}
```

Exploring package-java.io (file handling)

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −

Example

```
import java.io.*;
public class CopyFile {

  public static void main(String args[]) throws IOException {
    FileInputStream in = null;
    FileOutputStream out = null;

    try {
      in = new FileInputStream("input.txt");
      out = new FileOutputStream("output.txt");

      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    }finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
        out.close();
      }
    }
  }
}
```

Now let's have a file **input.txt** with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

Example

```
import java.io.*;
public class CopyFile {

  public static void main(String args[]) throws IOException {
    FileReader in = null;
    FileWriter out = null;
```

```
    try {
      in = new FileReader("input.txt");
      out = new FileWriter("output.txt");

      int c;
      while ((c = in.read()) != -1) {
        out.write(c);
      }
    }finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
        out.close();
      }
    }
  }
}
```

Now let's have a file **input.txt** with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

- **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
- **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "q" −

Example

```
import java.io.*;
public class ReadConsole {

  public static void main(String args[]) throws IOException {
    InputStreamReader cin = null;

    try {
      cin = new InputStreamReader(System.in);
```

```
        System.out.println("Enter characters, 'q' to quit.");
        char c;
        do {
          c = (char) cin.read();
          System.out.print(c);
        } while(c != 'q');
      }finally {
        if (cin != null) {
          cin.close();
        }
      }
    }
}
```

Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q' −

```
$javac ReadConsole.java
$java ReadConsole
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.

The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed in this tutorial.

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file −

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

Sr.No.	Method & Description

1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. T IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output called when there are no more references to this stream. Throws an IOException.
3	**public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the nex data and -1 will be returned if it's the end of the file.
4	**public int read(byte[] r) throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of byte it is the end of the file, -1 will be returned.
5	**public int available() throws IOException{}** Gives the number of bytes that can be read from this file input stream. Returns an int.

There are other important input streams available, for more detail you can refer to the following links −

- ByteArrayInputStream
- DataInputStream

FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file −

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. T

	IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output called when there are no more references to this stream. Throws an IOException.
3	**public void write(int w)throws IOException{}** This methods writes the specified byte to the output stream.
4	**public void write(byte[] w)** Writes w.length bytes from the mentioned byte array to the OutputStream.

There are other important output streams available, for more detail you can refer to the following links −

- ByteArrayOutputStream
- DataOutputStream

Example

Following is the example to demonstrate InputStream and OutputStream −

```
import java.io.*;
public class fileStreamTest {

  public static void main(String args[]) {

    try {
      byte bWrite [] = {11,21,3,40,5};
      OutputStream os = new FileOutputStream("test.txt");
      for(int x = 0; x < bWrite.length ; x++) {
        os.write( bWrite[x] );   // writes the bytes
      }
      os.close();

      InputStream is = new FileInputStream("test.txt");
      int size = is.available();

      for(int i = 0; i < size; i++) {
        System.out.print((char)is.read() + "  ");
      }
      is.close();
    } catch (IOException e) {
      System.out.print("Exception");
    }
  }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

Directories in Java

A directory is a File which can contain a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on File object and what are related to directories.

Creating Directories

There are two useful **File** utility methods, which can be used to create directories −

- The **mkdir( )** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory −

Example

```
import java.io.File;
public class CreateDir {

  public static void main(String args[]) {
    String dirname = "/tmp/user/java/bin";
    File d = new File(dirname);

    // Create directory now.
    d.mkdirs();
  }
}
```

Compile and execute the above code to create "/tmp/user/java/bin".

Note − Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Listing Directories

You can use **list( )** method provided by **File** object to list down all the files and directories available in a directory as follows −

Example

```
import java.io.File;
public class ReadDir {

  public static void main(String[] args) {
    File file = null;
    String[] paths;

    try {
      // create new file object
```

```
        file = new File("/tmp");

        // array of files and directory
        paths = file.list();

        // for each name in the path array
        for(String path:paths) {
          // prints filename and directory name
          System.out.println(path);
        }
      } catch (Exception e) {
        // if any error occurs
        e.printStackTrace();
      }
    }
}
```

This will produce the following result based on the directories and files available in your **/tmp** directory −

Output

```
test1.txt
test2.txt
ReadDir.java
ReadDir.class
```

Exception Handling:

Multithreading - Thread life cycle, creating threads, synchronizing threads, daemon threads.

Exception Handling

Definition

An exception is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. Java's exception handling brings run-time error management into the object-oriented world.

Advantages of exception handling:

- Exception handling enables you to create applications that can resolve (or handle) exceptions.
- This feature enables programmers to write robust and fault-tolerant programs (i.e., programs that are able to deal with problems that may arise and continue executing).

Exception-Handling Fundamentals

When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. At some point, the exception is caught and processed.

Exceptions are of two types in java:

- User defined exceptions
- Predefined exceptions

Java exception handling is managed via five keywords: try, catch, throw, throws, and finally. Program statements that are to be monitored for exceptions are contained within a try block.

- If an exception occurs within the try block, it is thrown.
- The exception can be caught (using catch) and handle it in some rational manner.
- 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.

This is the general form of an exception-handling block:

```
try {
      // block of code to monitor for errors
      //code that may contain a run time error
}catch (ExceptionType1 exOb) {
      // exception handler for ExceptionType1
}catch (ExceptionType2 exOb) {
      // exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed before try block ends
}
```

Here, ExceptionType is the type of exception that has occurred. At least one catch block or a finally block must immediately follow the try block. Each catch block specifies in parentheses an exception parameter that identifies the exception type the handler can process. When an exception occurs in a try block, the catch block that executes is the one whose type matches the type of the exception that occurred

Exception Types

All exception types are subclasses of the built-in class **Throwable**. 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 the 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.

Uncaught Exceptions

When an exception is not handled then what happens is explained using the run time error divide by zero.

```
import java.io.*;
import java.util.*;
class exc {
        public static void main(String args[])throws IOException {
                BufferedReader bf=new BufferedReader(new InputStreamReader(System.in));
                int a=Integer.parseInt(bf.readLine());
                 b=Integer.parseInt(bf.readLine());
                int c=a/b;
        System.out.println(c);
        }
}
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then throws this exception. The program terminates. Once an exception has been thrown, it must be caught by an exception handler. In this example, exception handler is not available. Any exception that is not caught by the program will ultimately be processed by the default handler. The default handler displays a string describing the exception.

Output:

```
java.lang.ArithmeticException: / by zero
at exc.main(exc.java:4)
```

Using try and catch

There are two benefits of handling an exception.

- First, it allows the programmer to fix the error.
- Second, it prevents the program from automatically terminating.

To handle a run-time error, simply enclose the code that is to be monitored inside a try block. Immediately following the try block, include a catch clause that specifies the exception type that is to be caught. ArithmeticException generated by the division-by-zero error:

```
import java.io.*;
import java.util.*;
```

```
class exctry {
        public static void main(String args[])throws IOException {
                BufferedReader bf=new BufferedReader(new InputStreamReader(System.in));
                int a=Integer.parseInt(bf.readLine());
                int b=Integer.parseInt(bf.readLine()); int c;
                try {
                        c=a/b;
                }catch(ArithmeticException e) {
                        System.out.println("Arithmetic- divide by zero exception"); c=1;
                }
                System.out.println(c);
        }
}
```

- Once an exception is thrown, program control transfers out of the try block into the catch block.
- Execution never “returns” to the try block from a catch.
- Once the catch statement has executed, program control continues with the next line in the program following the entire try/catch mechanism.
- The statements that are protected by try must be surrounded by curly braces.

Displaying a Description of an Exception

Description of the exception can be displayed with a println( ) statement by simply passing the exception as an argument. For example, the catch block in the following program can be rewritten like this:

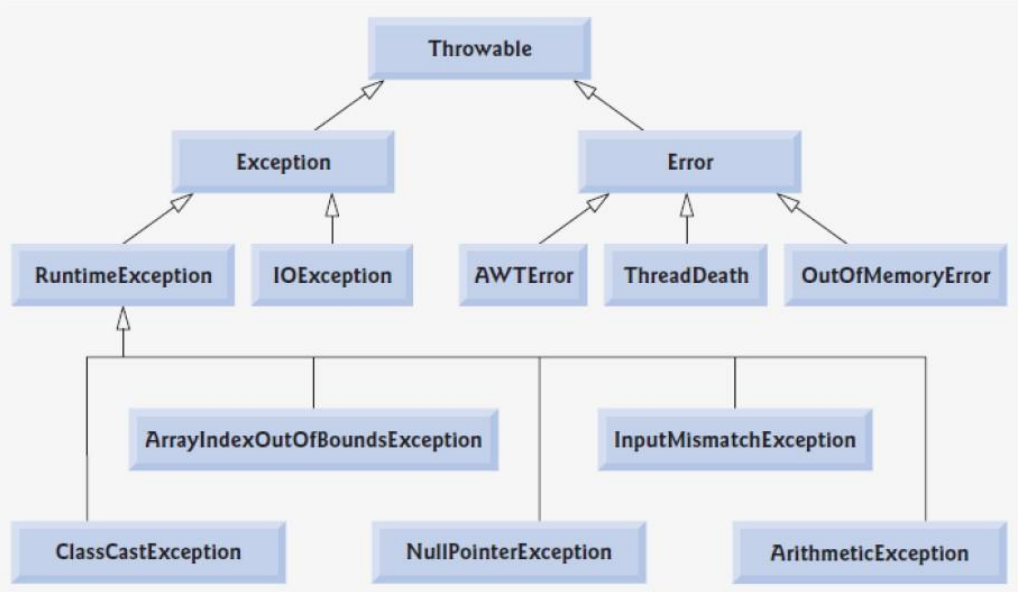
```
class exctry1 {
        public static void main(String args[]) {
                int a=10,b=0,c;
                try {
                        c=a/b;
                }catch(ArithmeticException e) {
                        System.out.println("Arithmetic- divide by zero exception"+e);
                }
        }
}
```

Output:

```
Exception: java.lang.ArithmeticException: / by zero
```

The Throwable Class and Its Subclasses

Java language requires exceptions derive from the Throwable class or one of its subclasses. Throwable is a subclass of the Object class. Throwable has two direct descendants: **Error** and **Exception**.



Errors

When a dynamic linking failure or some other "hard" failure in the virtual machine occurs, the virtual machine throws an Error. Typical Java programs should not catch Errors.

Exceptions

Most programs throw and catch objects that derive from the Exception class. The Exception class has many descendants defined in the Java packages. These descendants indicate various types of exceptions that can occur. For example, ArrayIndexOutOfBoundsException, ArithmeticException, IllegalAccessException, NegativeArraySizeException etc;

Runtime Exceptions

One Exception subclass has special meaning in the Java language: RuntimeException. The RuntimeException class represents exceptions that occur within the Java virtual machine (during runtime). An example of a runtime exception is NullPointerException, which occurs when a method tries to access a member of an object through a null reference. A NullPointerException can occur anywhere a program tries to dereference a reference to an object. The compiler allows runtime exceptions to go uncaught and unspecified.

Multiple catch Clauses

A try statement can be followed by more catch statements to handle more than one exception that could be raised by a single piece of code. Each catch clause catches 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. The others are bypassed, and execution continues after the try/catch block.

Example: To demonstrate multiple catch statements.

```
class multicatch {
    public static void main(String args[]) {
        int a,b,c; a=Integer.parseInt(args[0]);
        b=Integer.parseInt(args[1]);
```

```
                int x[]={1,2,3,4};
                try {
                        c=a/b;
                        System.out.println(“x[c]=”x[c]);
                }catch(ArithmeticException e) {
                        System.out.println("Arithmetic- divide by zero exception");
                }catch(ArrayIndexOutOfBoundsException e)        {
                        System.out.println("ArrayIndexOut of bounds");
                }
        }
}
```

Here is the output generated by running it different ways:

C:\>java multicatch 10 5 X[c]=3

C:\>java multicatch 10 2

Array index oob: java.lang.ArrayIndexOutOfBoundsException C:\>java multicatch 10 0

Arithmetic- divide by zero exception

Note: When multiple catch statements are used, 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.

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 run-time system will handle the exception. An example that uses nested try statements:

```
import java.io.*;
import java.util.*;
class nesttry {
        public static void main(String args[]) {
                int a,b,c;
                a=Integer.parseInt(args[0]);
                b=Integer.parseInt(args[1]);
                int x[]={1,2,3,4};
                try {
                        c=a/b;
                        try{
                                System.out.println(x[100]);
                        }catch(NullPointerException e)        {
                                System.out.println("ArrayIndexOut of bounds");
                        }
                }catch(ArithmeticException e) {
                        System.out.println("Arithmetic- divide by zero exception");
```

```
            }catch(ArrayIndexOutOfBoundsException e)        {
            System.out.println("ArrayIndexOut of bounds");
            }
        }
}
```

throw

In general exceptions are thrown by Java run-time system. They can be handled in the programs. Programmers can throw exceptions by using the throw statement. A throw statement specifies an object to be thrown.

The general form of throw is:

throw ThrowableInstance;

The operand of a throw can be of any class derived from class Throwable. The flow of execution stops immediately after the throw statement; any subsequent statements 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 and prints the stack trace.

Here is a sample program that creates and throws an exception.

```
class throwex {
        public static void main(String args[]) {
                try {
                        throw new NullPointerException("thow example");
                        //System.out.println("after throw");
                }catch(NullPointerException e) {
                        System.out.println("nullpointer exception"+e);
                }
        }
}
```

Rethrowing an exception

When a catch block receives an exception some times it can not process that exception or can only partially process it. Then the exceptions can be rethrown. Rethrowing an exception defers the exception handling to another catch block associated with an outer try statement. An exception is rethrown by using the throw keyword, followed by a reference to the exception object that was just caught. Note that exceptions cannot be rethrown from a finally block.

```
class rethrow {
        static void fun() {
                try {
                        throw new NullPointerException("demo");
                }catch(NullPointerException e) {
                        System.out.println("Caught inside fun");
                        throw e; // rethrow the exception
                }
        }
```

```
        public static void main(String args[]) {
                try {
                        fun();
                }
                catch(NullPointerException e) {
                        System.out.println("Recaught: " + e);
        }
}
}
```

This program gets two chances to deal with the same error. First, main( ) sets up an exception context and then calls fun( ). The fun( ) method then sets up another exception-handling context and immediately throws a new instance of NullPointerException, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside fun.
Recaught: java.lang.NullPointerException: demo
```

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. This can be accomplished by adding a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other 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 the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list {
// body of method
}
```

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

```
class ThrowsDemo {
        static void throwOne() throws IllegalAccessException {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
        }
        public static void main(String args[]) {
                try {
                        throwOne();
                }catch (IllegalAccessException e) {
                        System.out.println("Caught " + e);
                }
        }
}
```

finally

finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception. The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

Java's Built-in Exceptions (checked and unchecked exceptions):

Java defines several exception classes in java.lang package. The most general of these exceptions are subclasses of the standard type **RuntimeException**. Since java.lang is implicitly imported into all Java programs, most exceptions derived from RuntimeException are automatically available. Furthermore, they need not be included in any method's throws list. unchecked exceptions are the exceptions, the compiler does not check to see if a method handles or throws these exceptions. Some of the unchecked exceptions defined in java.lang are

Exception	Meaning
ArithmeticException	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.
NegativeArraySizeException	Array created with a negative size
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion

Checked exceptions: exceptions defined by java.lang that must be included in a method's throws list if that method can generate any exception and does not handle it itself. These are called checked exceptions. Following are some of the checked exceptions in java.lang.

Exception	Meaning
ClassNotFoundException	Class not found
IllegalAccessException	Access to a class is denied.
NoSuchFieldException	A requested field does not exist..
NoSuchMethodException	A requested method does not exist

Creating Own Exception Subclasses

In addition to the predefined exception classes programmer can create his/her own exceptions which are suitable for specific applications. These are called as user defined exceptions. To accomplish this a class must be defined as a subclass of Exception class.

Example

```
import java.io.*;
import java.util.*;
 class InvalidNumber extends Exception {
        int k;
        InvalidNumber(int k) {
                this.k=k;
        }
        public String toString() {
```

```
                return ("number must be < 100, you have entered"+k);
        }
}
class myexc {
        public static void main(String args[])throws IOException {
                BufferedReader bf=new BufferedReader(new
                InputStreamReader(System.in)); System.out.println("Enter a number");
                int
                a=Integer.parseInt(bf.readLine());
                try     {
                        if(a>100)
                                throw new InvalidNumber(a);
                                System.out.println("a value
                                is:"+a);
                }catch(InvalidNumber e)     {
                        System.out.println("caught:"+
                        e);
                }
        }
}
```

This example defines a subclass of Exception called InvalidNumber. This subclass is quite simple: it has only a constructor plus an overloaded toString( ) method that displays the value of the exception. The myexc class defines the main method that throws a InvalidNumber object. The exception is thrown when user enters a value greater than 100 as input.

Chained Exceptions

The chained exception feature allows you 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, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

Throwable(Throwable causeExc)
Throwable(String msg, Throwable causeExc)

Module-4

Multithreading – Differences between multithreading and Multitasking,Thread life cycle, creating threads, synchronizing threads, daemon threads,thread groups.

Collectionclasses: –

ArrayList,LinkedList,HashSet,LinkedHashSet,TreeSet,PriorityQueue,ArrayDequeue,EnumSet

Multithreading vs Multitasking

Multi-threading	Multitasking
It is a programming language concept in which a program or process is divided into two or more subprograms that are executed at the same time in parallel.	It is an operating system concept in which multiple tasks are performed simultaneously.
It is highly efficient.	It is less efficient than Multi-threading.
It supports execution of multiple parts of a single program simultaneously.	It supports execution of multiple programs simultaneously.
The processor has to switch between different parts or threads of a program.	The processor has to switch between different programs or processes.
A thread is the smallest unit in multi-threading.	A program or process is the smallest unit in multitasking.
It is cost effective in case of context switching.	It is expensive in case of context switching.
It helps in developing efficient programs.	It helps in developing efficient operating systems.

MultiThreading

Java provides built-in support for multithreaded programming. A thread is a path of execution in a program. A multithreaded program contains two or more parts such threads that can run concurrently.

There are two distinct types of multitasking:

- process-based and
- thread based.

Process based multi-tasking

A process is a program that is executing. Thus, process-based multitasking is the feature that allows to run two or more programs concurrently. For example, process-based multitasking enables to run the Java compiler while using a text editor. In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler. Processes are heavyweight tasks that require their own separate address spaces. Interprocess communication is expensive and limited. Context switching from one process to another is also costly.

Thread based multi-tasking

A thread is a path of execution in a program. This means that a single program can perform two or more tasks simultaneously. For instance, a text editor can format text at the same time that it is printing, as long as

these two actions are being performed by two separate threads. Multitasking threads require less overhead than multitasking processes. Threads, on the other hand, are lightweight. They share the same address space. Interthread communication is inexpensive, and context switching from one thread to the next is low cost.

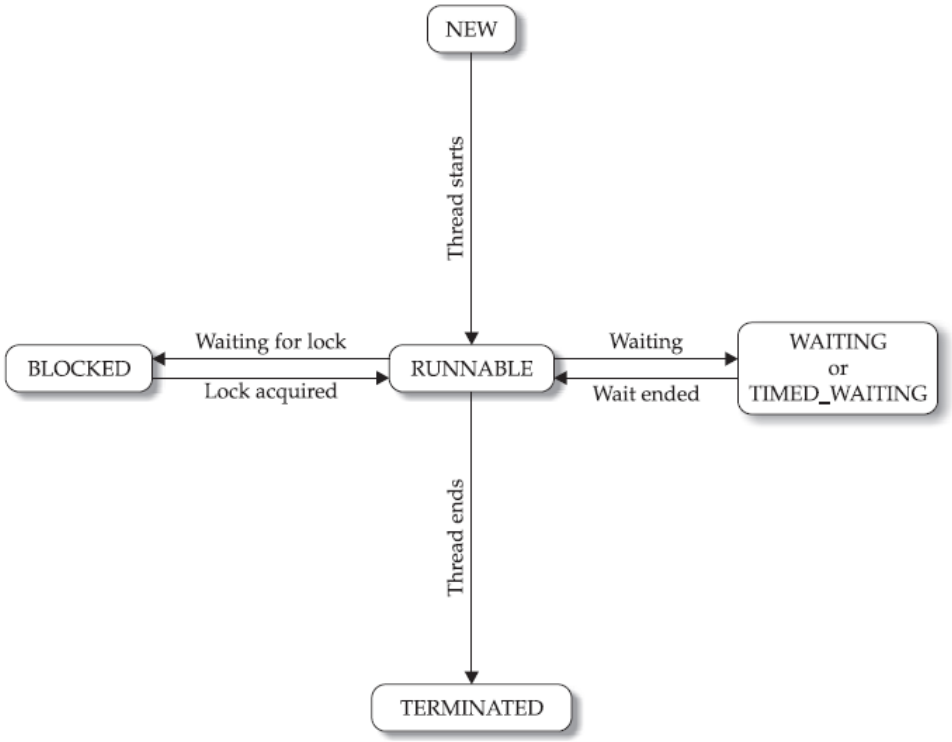
Advantages

Multithreading enables to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum. Multithreading gain access to the idle time and put the CPU to good use.

Thread States (or) Thread Life cycle

The following diagram shows the life cycle of a Java Thread. Thread has following states: Start, Stop, Blocked, Running, Runnable. These states can be changed to each other as shown in the diagram. Thread can sleep, notify, resume, start, stop, suspend, yield and wait. These activities lead the thread from one state to move to another one.

1. **New state** – After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.
2. **Runnable (Ready-to-run) state** – A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
3. **Running state –** A thread is in running state that means the thread is currently executing.
4. **Dead state –** A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
5. **Blocked** - A thread can enter in this state because of waiting the resources that are hold by another thread.



The Thread Class and main thread:

Java's multithreading system is built upon the Thread class. Thread encapsulates a thread of execution. TheThread class defines several methods that help manage threads.

Method	Meaning
getName ()	Obtain a thread's name.

getPriority()	Obtain a thread's priority.
isAlive()	Determine if a thread is still running.
Join()	Wait for a thread to terminate.
run ()	Entry point for the thread.
sleep ()	Suspend a thread for a period of time.
start ()	Start a thread by calling its run method.

When a Java program starts up, one thread begins running immediately. This is usually called the main thread of the program The main thread can be controlled through a Thread object. To do so, you must obtain a reference to it by calling the method currentThread( ). Its general form is shown here:

Thread currentThread( )

This method returns a reference to the thread in which it is called.

Example

```
// Controlling the main Thread.
class DemoThread {
        public static void main(String args[]) {
                Thread t = Thread.currentThread();
                System.out.println("Current thread: " + t);
                t.setName("My Thread");
                System.out.println("After name change: " + t);
                try {
                        for(int n = 5; n > 0; n--) {
                                System.out.println(n);
                                Thread.sleep(1000);
                        }
                }catch (InterruptedException e)         {
                        System.out.println("Main thread interrupted");
                }
        }
}
```

Sleep()

The sleep( ) method causes the thread from which it is called to suspend execution for the specified period of milliseconds. The number of milliseconds to suspend is specified in milliseconds. This method may throw an InterruptedException. Its general form is:

static void sleep(long milliseconds) throws InterruptedException

Creating a Thread

Java defines two ways to create a thread

- By implementing the Runnable interface.
- By extending the Thread class, itself.

a. Implementing Runnable

The easiest way to create a thread is to create a class that implements the Runnable. interface. Following are the steps to create a thread by implementing Runnable interface.

1. Define a class that implements Runnable interface
2. Implement the method run().
3. Create a thread by passing an object of this runnable class to the Thread class constructor
4. Call the thread's start() method to run the thread. Example:

```
import java.io.*;
class myThread implements Runnable {
        public void run() {
```

```
                System.out.println("Run method");
        }
}
class runthreadex {
        public static void main(String args[]) {
                myThread m=new myThread();
                Thread t=new Thread(m);
                t.start();
        }
}
```

Inside run( ), you will define the code that constitutes the new thread. After creating a class that implements Runnable, instantiate an object of that class. Now create an object of type Thread by passing the above object as parameter to the Thread constructor. The start( ) executes a call to run( ). The start( ) method is shown here:

void start( )

Here is an example that creates a new thread and starts it running:

```
// Create a second thread.
class NewThread implements Runnable {
        Thread t;
        NewThread() {
                // Create a new, second thread
                t = new Thread(this, "Demo Thread");
                System.out.println("Child thread: " + t);
                t.start(); // Start the thread
        }
        // This is the entry point for the second thread.
        public void run() {
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Child Thread: " + i);
                                Thread.sleep(500);
                        }
                } catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
                }
                        System.out.println("Exiting child thread.");
                }
        }
class ThreadDemo {
        public static void main(String args[]) {
                 new NewThread(); // create a new thread
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Main Thread: " + i);
                                Thread.sleep(1000);
                        }
                } catch (InterruptedException e) {
                        System.out.println("Main thread interrupted.");
                }
                System.out.println("Main thread exiting.");
```

```
        }
}
```

Inside NewThread's constructor, a new Thread object is created by the following statement:

t = new Thread(this, "Demo Thread");

Passing this as the first argument indicates that you want the new thread to call the run( ) method on this object. Next, start( ) is called, which starts the thread of execution beginning at the run( ) method. This causes the child thread's for loop to begin. After calling start( ), NewThread's constructor returns to main( ). When the main thread resumes, it enters its for loop. Both threads continue running, sharing the CPU, until their loops finish. The output produced by this program is as follows:

Child thread: Thread[Demo Thread,5,main] Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1 Exiting child thread. Main Thread: 2
Main Thread: 1 Main thread exiting.

As mentioned earlier, in a multithreaded program, often the main thread must be the last thread to finish running. In fact, for some older JVMs, if the main thread finishes before a child thread has completed, then the Java run- time system may"hang." The preceding program ensures that the main thread finishes last, because the main thread sleeps for 1,000 milliseconds between iterations, but the child thread sleeps for only 500 milliseconds. This causes the child thread to terminate earlier than the main thread. Shortly, you will see a better way to wait for a thread to finish.

Extending Thread

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. Here is the preceding program rewritten to extend Thread:

```
// Create a second thread by extending Thread
class NewThread extends Thread {
        NewThread() {
                // Create a new, second thread
                super("Demo Thread");
                System.out.println("Child thread: " + this);
                start(); // Start the thread
        }
        // This is the entry point for the second thread.
        public void run() {
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Child Thread: " + i);
                                Thread.sleep(500);
                        }
                } catch (InterruptedException e) {
                        System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
}
}
```

```
class ExtendThread {
        public static void main(String args[]) {
                new NewThread(); // create a new thread
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println("Main Thread: " + i);
                                Thread.sleep(1000);
                        }
                } catch (InterruptedException e) {
                        System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
}
}
```

This program generates the same output as the preceding version. As you can see, the child thread is created by instantiating an object of NewThread, which is derived from Thread.

Notice the call to super( ) inside NewThread. This invokes the following form of the Thread constructor:

public Thread(String threadName)

Here, threadName specifies the name of the thread. Choosing an Approach

At this point, you might be wondering why Java has two ways to create child threads, and which approach is better. The answers to these questions turn on the same point. The Thread class defines several methods that can be overridden by a derived class. Of these methods, the only one that must be overridden is run( ). This is, of course, the same method required when you implement Runnable. Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way. So, if you will not be overriding any of Thread's other methods, it isprobably best simply to implement Runnable. This is up to you, of course. However, throughout the rest of this chapter, we will create threads by using classes that implement Runnable.

Creating Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, your program can spawn as many threads as it needs. For example, the following program creates three child threads:

```
// Create multiple threads.
class NewThread implements Runnable {
        String name; // name of thread
        Thread t;
        NewThread(String threadname) {
                name = threadname;
                t = new Thread(this, name);
                System.out.println("New thread: " + t);
                t.start(); // Start the thread
        }
 // This is the entry point for thread.
        public void run() {
                try {
                        for(int i = 5; i > 0; i--) {
                                System.out.println(name + ": " + i);
                                Thread.sleep(1000);
                        }
                } catch (InterruptedException e) {
                        System.out.println(name + "Interrupted");
                }
```

```
        System.out.println(name + " exiting.");
}
}
class MultiThreadDemo {
        public static void main(String args[]) {
                new NewThread("One"); // start threads
                new NewThread("Two");
                new NewThread("Three");
                try {
                        // wait for other threads to end
                        Thread.sleep(10000);
                } catch (InterruptedException e) {
                        System.out.println("Main thread Interrupted");
                }
                System.out.println("Main thread exiting.");
        }
}
```

The output from this program is shown here:

```
New thread: Thread[One,5,main] New thread: Thread[Two,5,main] New thread: Thread[Three,5,main] One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting. Two exiting. Three exiting.
Main thread exiting.
```

As you can see, once started, all three child threads share the CPU. Notice the call to sleep(10000) in main( ). This causes the main thread to sleep for ten seconds and ensures that it will finish last.

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. As you will see, Java provides unique, language-level support for it.

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, or mutex. 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.

If you have worked with synchronization when using other languages, such as C or C++, you know that it can be a bit tricky to use. This is because most languages do not, themselves, support synchronization. Instead, to synchronize threads, your programs need to utilize operating system primitives. Fortunately, because Java implements synchronization through language elements, most of the complexity associated with

synchronization has been eliminated.

You can synchronize your code in either of two ways. Both involve the use of the synchronized keyword, and both are examined here.

Using Synchronized Methods

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.

To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, Callme, has a single method named call( ). The call( ) method takes a String parameter called msg. This method tries to print the msg string inside of square brackets. The interesting thing to notice is that after call( ) prints the opening bracket and the msg string, it calls Thread.sleep(1000), which pauses the current thread for one second.

The constructor of the next class, Caller, takes a reference to an instance of the Callme class and a String, which are stored in target and msg, respectively. The constructor also creates a new thread that will call this object's run( ) method. Thethread is started immediately. The run( ) method of Caller calls the call( ) method on the target instance of Callme, passing in the msg string. Finally, the Synch class starts by creating a single instance of Callme, and three instances of Caller, each with a unique message string. The same instance of Callme is passed to each Caller.

```
// This program is not synchronized.
class Callme {
	void call(String msg) {
		System.out.print("[" + msg);
		try {
			Thread.sleep(1000);
		} catch(InterruptedException e) {
			System.out.println("Interrupted");
		}
		System.out.println("]");
	}
}
class Caller implements Runnable {
	String msg;
	Callme target;
	Thread t;
	public Caller(Callme targ, String s) {
		target = targ;
		msg = s;
		t = new Thread(this);
		t.start();
	}
	public void run() {
		target.call(msg);
	}
}
class Synch {
	public static void main(String args[]) {
		Callme target = new Callme();
		Caller ob1 = new Caller(target, "Hello");
```

```
                Caller ob2 = new Caller(target, "Synchronized");
                Caller ob3 = new Caller(target, "World");
                // wait for threads to end
                try {
                        ob1.t.join();
                        ob2.t.join();
                        ob3.t.join();
                } catch(InterruptedException e) {
                        System.out.println("Interrupted");
                }
        }
}
```

Output:

```
Hello[Synchronized[World
]
]
```

As you 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 the three threads are racing each other to complete the method. This example used sleep( ) to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right one time and wrong the next. To fix the preceding program, you must serialize access to call( ). That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede call( )'s definition with the keyword synchronized, as shown here:

```
class Callme {
        synchronized void call(String msg) {
                ...
        }
}
```

This prevents other threads from entering call( ) while another thread is using it. After synchronized has been added to call( ), the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the synchronized keyword to guard the state from race conditions. Remember, 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.

The synchronized Statement:

While creating synchronized methods within classes that you 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 you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use

synchronized methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add synchronized to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a synchronized block.

This is the general form of the synchronized statement:

```
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 after the current thread has successfully entered object's monitor.

Here is an alternative version of the preceding example, using a synchronized block within the run( ) method:

```
// This program uses a synchronized block.

class Callme {
        void call(String msg) {
                 System.out.print("[" + msg);
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e) {
                        System.out.println("Interrupted");
                }
                System.out.println("]");
        }
}
class Caller implements Runnable {
        String msg;
        Callme target;
        Thread t;
        public Caller(Callme targ, String s) {
                target = targ;
                msg = s;
                t = new Thread(this);
                t.start();
        }
        // synchronize calls to call()
        public void run() {
                synchronized(target) { // synchronized block
                        target.call(msg);
                }
        }
}
class Synch1 {
        public static void main(String args[]) {
                Callme target = new Callme();
                Caller ob1 = new Caller(target, "Hello");
                Caller ob2 = new Caller(target, "Synchronized");
                Caller ob3 = new Caller(target, "World");
                // wait for threads to end
                try {
                        ob1.t.join();
                        ob2.t.join();
                        ob3.t.join();
                } catch(InterruptedException e) {
                        System.out.println("Interrupted");
                }
```

```
        }
}
```

Here, the call( ) method is not modified by synchronized. Instead, the synchronized statement is used inside Caller's run( ) method. This causes the same correct output as the preceding example, because each thread waits for the prior one to finish before proceeding.

Interthread Communication

The preceding examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through interprocess communication. As you will see, this is especially easy in Java.

Daemon Thread

It is a service provider thread that provides services to the user thread. Its life depends on the mercy of the user threads i.e. when all the user threads dies, JVM terminates this thread automatically.

- We have gc, finalizer etc. as daemon threads in java.
- We have the following 2 methods on this.

1. public void setDaemon(boolean status) : is used to mark the current thread as daemon thread
2. public boolean isDaemon() : is used to check that current thread is daemon.

Daemon Thread Example Programs:

```
public class DaemonThread extends Thread{
        public void run(){
                if(Thread.currentThread().isDaemon())
                        System.out.println("daemon thread work");
         else
                        System.out.println("user thread work");
        }
        public static void main(String[] args){
              DaemonThread t1=new DaemonThread();
              DaemonThread t2=new DaemonThread();
              t1.setDaemon(true);//now t1 is daemon thread
             t1.start();    t2.start();
         }
}
```

```
C:\java>javac DaemonThread.java

C:\java>java DaemonThread
daemon thread work
user thread work
```

```
public class DaemonThread extends Thread{
        public void run(){
                if(Thread.currentThread().isDaemon())
                 System.out.println("daemon thread work");
                else
                System.out.println("user thread work");
        }
         public static void main(String[] args){
        DaemonThread t1=new DaemonThread();
         t1.setDaemon(true);//now t1 is daemon thread
         t1.start();
         }
}
```

```
C:\java>javac DaemonThread.java
C:\java>java DaemonThread
```

Thread groups

Java provides a convenient way to group multiple threads in a single object. In such a way, we can suspend, resume or interrupt a group of threads by a single method call.

Java thread group is implemented by *java.lang.ThreadGroup* class.

A ThreadGroup represents a set of threads. A thread group can also include the other thread group. The thread group creates a tree in which every thread group except the initial thread group has a parent.

A thread is allowed to access information about its own thread group, but it cannot access the information about its thread group's parent thread group or any other thread groups.

Constructors of ThreadGroup class

There are only two constructors of ThreadGroup class.

No.	Constructor	Description
1)	ThreadGroup(String name)	creates a thread group with given name.
2)	ThreadGroup(ThreadGroup parent, String name)	creates a thread group with a given parent group and name.

Methods of ThreadGroup class

There are many methods in ThreadGroup class. A list of ThreadGroup methods is given below.

S.N.	Modifier and Type	Method	Description
1)	void	checkAccess()	This method determines if the currently running thread has permission to modify the thread group.
2)	int	activeCount()	This method returns an estimate of the number of active threads in the thread group and its subgroups.
3)	int	activeGroupCount()	This method returns an estimate of the number of

			active groups in the thread group and its subgroups.
4)	void	destroy()	This method destroys the thread group and all of its subgroups.
5)	int	enumerate(Thread[] list)	This method copies into the specified array every active thread in the thread group and its subgroups.
6)	int	getMaxPriority()	This method returns the maximum priority of the thread group.
7)	String	getName()	This method returns the name of the thread group.
8)	ThreadGroup	getParent()	This method returns the parent of the thread group.
9)	void	interrupt()	This method interrupts all threads in the thread group.
10)	boolean	isDaemon()	This method tests if the thread group is a daemon thread group.
11)	void	setDaemon(boolean daemon)	This method changes the daemon status of the thread group.
12)	boolean	isDestroyed()	This method tests if this thread group has been destroyed.
13)	void	list()	This method prints information about the thread group to the standard output.
14)	boolean	parentOf(ThreadGroup g	This method tests if the thread group is either the thread group argument or one of its ancestor thread groups.
15)	void	suspend()	This method is used to suspend all threads in the thread group.
16)	void	resume()	This method is used to resume all threads in the thread group which was suspended using suspend() method.
17)	void	setMaxPriority(int pri)	This method sets the maximum priority of the group.
18)	void	stop()	This method is used to stop all threads in the thread group.
19)	String	toString()	This method returns a string representation of the Thread group.

Let's see a code to group multiple threads.

1. ThreadGroup tg1 = **new** ThreadGroup("Group A");
2. Thread t1 = **new** Thread(tg1,**new** MyRunnable(),"one");
3. Thread t2 = **new** Thread(tg1,**new** MyRunnable(),"two");
4. Thread t3 = **new** Thread(tg1,**new** MyRunnable(),"three");

Now all 3 threads belong to one group. Here, tg1 is the thread group name, MyRunnable is the class that implements Runnable interface and "one", "two" and "three" are the thread names.

Now we can interrupt all threads by a single line of code only.

1. Thread.currentThread().getThreadGroup().interrupt();

ThreadGroup Example

File: ThreadGroupDemo.java

```
public class ThreadGroupDemo implements Runnable{
 public void run() {
        System.out.println(Thread.currentThread().getName());
   }
  public static void main(String[] args) {
    ThreadGroupDemo runnable = new ThreadGroupDemo();
       ThreadGroup tg1 = new ThreadGroup("Parent ThreadGroup");

       Thread t1 = new Thread(tg1, runnable,"one");
       t1.start();
       Thread t2 = new Thread(tg1, runnable,"two");
       t2.start();
       Thread t3 = new Thread(tg1, runnable,"three");
       t3.start();

       System.out.println("Thread Group Name: "+tg1.getName());
      tg1.list();

   }
}
```

Output:

one

two

three

Thread Group Name: Parent ThreadGroup

java.lang.ThreadGroup[name=Parent ThreadGroup,maxpri=10]

ArrayList

The ArrayList class implements the List interface. It uses a dynamic array to store the duplicate element of different data types. The ArrayList class maintains the insertion order and is non-synchronized. The elements stored in the ArrayList class can be randomly accessed. Consider the following example.

```
import java.util.*;
class TestJavaCollection1{
public static void main(String args[]){
ArrayList<String> list=new ArrayList<String>();//Creating arraylist
list.add("Ravi");//Adding object in arraylist
list.add("Vijay");
list.add("Ravi");
list.add("Ajay");
//Traversing list through Iterator
Iterator itr=list.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

Ravi

Vijay

Ravi

Ajay

Aja

LinkedList

LinkedList implements the Collection interface. It uses a doubly linked list internally to store the elements. It can store the duplicate elements. It maintains the insertion order and is not synchronized. In LinkedList, the manipulation is fast because no shifting is required.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection2{
```

```
public static void main(String args[]){
LinkedList<String> al=new LinkedList<String>();
al.add("Ravi");
al.add("Vijay");
al.add("Ravi");
al.add("Ajay");
Iterator<String> itr=al.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

Ravi

Vijay

Ravi

Ajay

HashSet

HashSet class implements Set Interface. It represents the collection that uses a hash table for storage. Hashing is used to store the elements in the HashSet. It contains unique items.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection7{
public static void main(String args[]){
//Creating HashSet and adding elements
HashSet<String> set=new HashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//Traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

Vijay

Ravi

Ajay

LinkedHashSet

LinkedHashSet class represents the LinkedList implementation of Set Interface. It extends the HashSet class and implements Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection8{
public static void main(String args[]){
LinkedHashSet<String> set=new LinkedHashSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
}
}
```

Output:

Ravi

Vijay

Ajay

TreeSet

Java TreeSet class implements the Set interface that uses a tree for storage. Like HashSet, TreeSet also contains unique elements. However, the access and retrieval time of TreeSet is quite fast. The elements in TreeSet stored in ascending order.

Consider the following example:

```
import java.util.*;
public class TestJavaCollection9{
public static void main(String args[]){
//Creating and adding elements
TreeSet<String> set=new TreeSet<String>();
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
//traversing elements
Iterator<String> itr=set.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
```

```
}
}
}
```

Output

Ajay

Ravi

Vijay

PriorityQueue

The PriorityQueue class implements the Queue interface. It holds the elements or objects which are to be processed by their priorities. PriorityQueue doesn't allow null values to be stored in the queue.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection5{
public static void main(String args[]){
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit Sharma");
queue.add("Vijay Raj");
queue.add("JaiShankar");
queue.add("Raj");
System.out.println("head:"+queue.element());
System.out.println("head:"+queue.peek());
System.out.println("iterating the queue elements:");
Iterator itr=queue.iterator();
while(itr.hasNext()){
System.out.println(itr.next());
}
queue.remove();
queue.poll();
System.out.println("after removing two elements:");
Iterator<String> itr2=queue.iterator();
while(itr2.hasNext()){
System.out.println(itr2.next());
}
}
}
```

Output:

```
head:Amit Sharma
head:Amit Sharma
iterating the queue elements:
Amit Sharma
Raj
JaiShankar
Vijay Raj
```

after removing two elements:
Raj
Vijay Raj

Deque Interface

Deque interface extends the Queue interface. In Deque, we can remove and add the elements from both the side. Deque stands for a double-ended queue which enables us to perform the operations at both the ends.

Deque can be instantiated as:

Deque d = **new** ArrayDeque();

ArrayDeque

ArrayDeque class implements the Deque interface. It facilitates us to use the Deque. Unlike queue, we can add or delete the elements from both the ends.

ArrayDeque is faster than ArrayList and Stack and has no capacity restrictions.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection6{
public static void main(String[] args) {
//Creating Deque and adding elements
Deque<String> deque = new ArrayDeque<String>();
deque.add("Gautam");
deque.add("Karan");
deque.add("Ajay");
//Traversing elements
for (String str : deque) {
System.out.println(str);
}
}
}
```

Output:
Gautam
Karan
Ajay

EnumSet

Java EnumSet class is the specialized Set implementation for use with enum types. It inherits AbstractSet class and implements the Set interface.

EnumSet class declaration

Let's see the declaration for java.util.EnumSet class.

public abstract class EnumSet<E **extends** Enum<E>> **extends** AbstractSet<E> **implements** Clon eable, Serializable

Methods of Java EnumSet class

Method	Description
static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType)	It is used to create an enum set containing all of the elements in the specified element type.
static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)	It is used to create an enum set initialized from the specified collection.
static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType)	It is used to create an empty enum set with the specified element type.
static <E extends Enum<E>> EnumSet<E> of(E e)	It is used to create an enum set initially containing the specified element.
static <E extends Enum<E>> EnumSet<E> range(E from, E to)	It is used to create an enum set initially containing the specified elements.
EnumSet<E> clone()	It is used to return a copy of this set.

Java EnumSet Example

```
import java.util.*;
enum days {
 SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumSetExample {
  public static void main(String[] args) {
    Set<days> set = EnumSet.of(days.TUESDAY, days.WEDNESDAY);
    // Traversing elements
    Iterator<days> iter = set.iterator();
    while (iter.hasNext())
```

```
    System.out.println(iter.next());
  }
}
```

Output:
TUESDAY
WEDNESDAY

Java EnumSet Example: allOf() and noneOf()

```
import java.util.*;
enum days {
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
public class EnumSetExample {
  public static void main(String[] args) {
    Set<days> set1 = EnumSet.allOf(days.class);
      System.out.println("Week Days:"+set1);
      Set<days> set2 = EnumSet.noneOf(days.class);
     System.out.println("Week Days:"+set2);
       }
}
```

Output:
Week Days:[SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY]
Week Days:[]

UNIT-5

Event Handling

Event and Listener (Java Event Handling)

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. The java.awt.event package provides many event classes and Listener interfaces for event handling.

Types of Event

The events can be broadly classified into two categories:

- **Foreground Events** - Those events which require the direct interaction of user.They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page etc.

Background Events - Those events that require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

Event Handling

Event Handling is the mechanism that controls the event and decides what should happen if an event occurs. This mechanism have the code which is known as event handler that is executed when an event occurs. Java Uses the Delegation Event Model to handle the events. This model defines the standard mechanism to generate and handle the events.Let's have a brief introduction to this model.

The Delegation Event Model has the following key participants namely:

- **Source** - The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.

- **Listener** - It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.

Event classes and Listener interfaces:

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform Event Handling

Following steps are required to perform event handling:

1. Implement the Listener interface and overrides its methods
2. Register the component with the Listener

For registering the component with the Listener, many classes provide the registration methods. For example:

- **Button**
  - public void addActionListener(ActionListener a){}
- **MenuItem**
  - public void addActionListener(ActionListener a){}
- **TextField**
  - public void addActionListener(ActionListener a){}
  - public void addTextListener(TextListener a){}
- **TextArea**
  - public void addTextListener(TextListener a){}
- **Checkbox**
  - public void addItemListener(ItemListener a){}
- **Choice**

- public void addItemListener(ItemListener a){}

- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

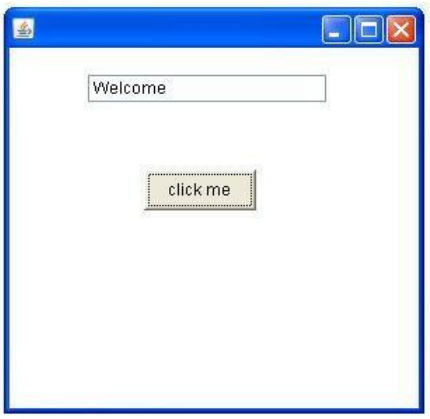
EventHandling Codes:

We can put the event handling code into one of the following places:

1. Same class
2. Other class
3. Annonymous class

Example of event handling within class:

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
TextField tf;
AEvent(){
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);
b.addActionListener(this);
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new AEvent();
} }
```

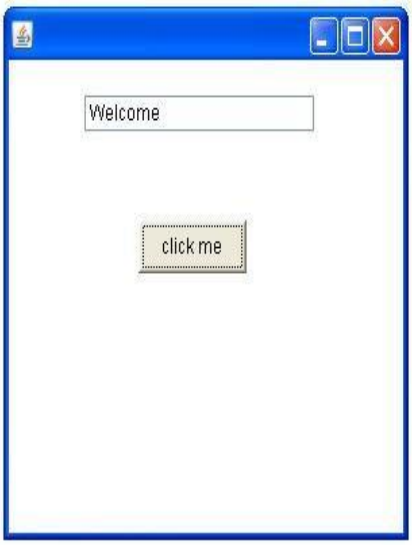


public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.

Java event handling by implementing ActionListener

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener{
```

```
TextField tf;
AEvent(){
//create components
tf=new TextField();
tf.setBounds(60,50,170,20);
Button b=new Button("click me");
b.setBounds(100,120,80,30);
//register listener
b.addActionListener(this);//passing current instance
//add components and set size, layout and visibility
add(b);add(tf);
setSize(300,300);
setLayout(null);
setVisible(true);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
}
public static void main(String args[]){
new AEvent();
}}
```



Java MouseListener Interface

The Java MouseListener is notified whenever you change the state of mouse. It is notified against MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods.

Methods of MouseListener interface

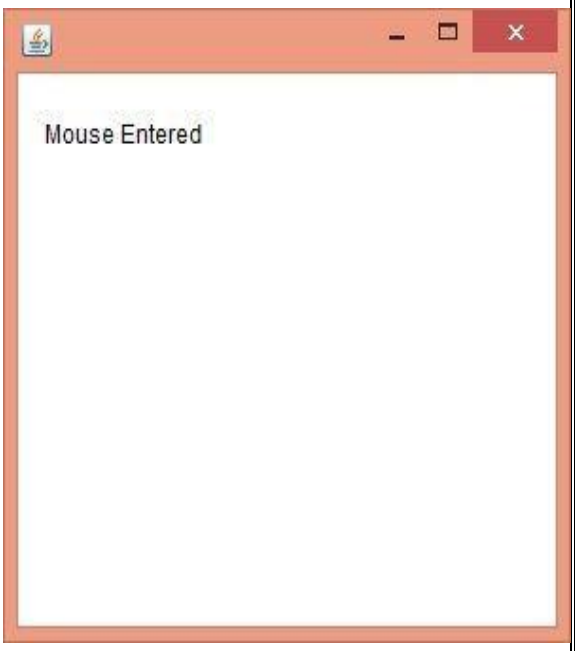
The signature of 5 methods found in MouseListener interface are given below:

1. public abstract void mouseClicked(MouseEvent e);
2. public abstract void mouseEntered(MouseEvent e);
3. public abstract void mouseExited(MouseEvent e);
4. public abstract void mousePressed(MouseEvent e);
5. public abstract void mouseReleased(MouseEvent e);

Java MouseListener Example

```
import java.awt.*;
import java.awt.event.*;
public class MouseListenerExample extends Frame implements MouseListener{
   Label l;
   MouseListenerExample(){
      addMouseListener(this);
      l=new Label();
      l.setBounds(20,50,100,20);
      add(l);
      setSize(300,300);
      setLayout(null);
      setVisible(true);
```

```
    }
    public void mouseClicked(MouseEvent e) {
        l.setText("Mouse Clicked");
    }
    public void mouseEntered(MouseEvent e) {
        l.setText("Mouse Entered");
    }
    public void mouseExited(MouseEvent e) {
        l.setText("Mouse Exited");
    }
    public void mousePressed(MouseEvent e) {
        l.setText("Mouse Pressed");
    }
    public void mouseReleased(MouseEvent e) {
        l.setText("Mouse Released");
    }
public static void main(String[] args) {
    new MouseListenerExample();
}}
```



Java KeyListener Interface

The Java KeyListener is notified whenever you change the state of key. It is notified against KeyEvent. The KeyListener interface is found in java.awt.event package. It has three methods.

Methods of KeyListener interface

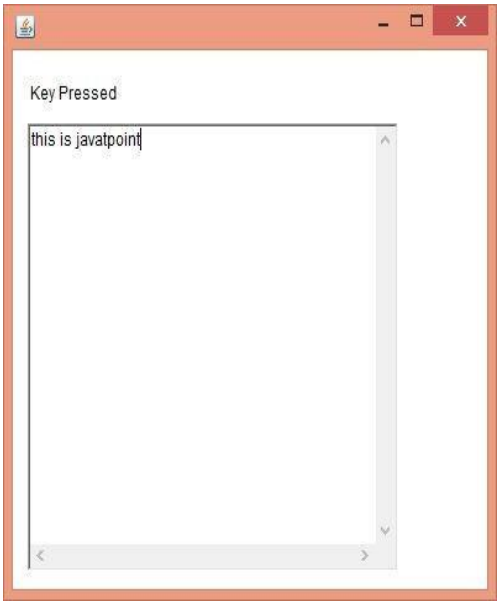
The signature of 3 methods found in KeyListener interface are given below:

1. public abstract void keyPressed(KeyEvent e);
2. public abstract void keyReleased(KeyEvent e);
3. public abstract void keyTyped(KeyEvent e);

Java KeyListener Example

```
import java.awt.*;
import java.awt.event.*;
public class KeyListenerExample extends Frame implements KeyListener{
    Label l;
    TextArea area;
    KeyListenerExample(){
        l=new Label();
        l.setBounds(20,50,100,20);
        area=new TextArea();
        area.setBounds(20,80,300, 300);
        area.addKeyListener(this);
        add(l);add(area);
        setSize(400,400);
        setLayout(null);
        setVisible(true);
```

```
}
public void keyPressed(KeyEvent e) {
   l.setText("Key Pressed");
}
public void keyReleased(KeyEvent e) {
   l.setText("Key Released");
}
public void keyTyped(KeyEvent e) {
   l.setText("Key Typed");
}
public static void main(String[] args) {
   new KeyListenerExample(); }  }
```



Java Adapter Classes

Java adapter classes *provide the default implementation of listener interfaces*. If you inherit the adapter class, you will not be forced to provide the implementation of all the methods of listener interfaces. So it *saves code*.

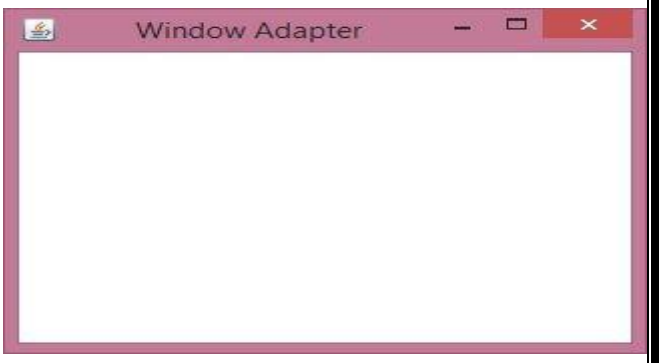
java.awt.event Adapter classes

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

Java WindowAdapter Example

```
1. import java.awt.*;
import java.awt.event.*;
public class AdapterExample{
```

```
    Frame  f;
    AdapterExample(){
        f=new Frame("Window Adapter");
        f.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent e) {
                f.dispose();     }    });
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
    }
public static void main(String[] args) {
    new AdapterExample();
} }
```



GUI Programming with java

The AWT Class hierarchy

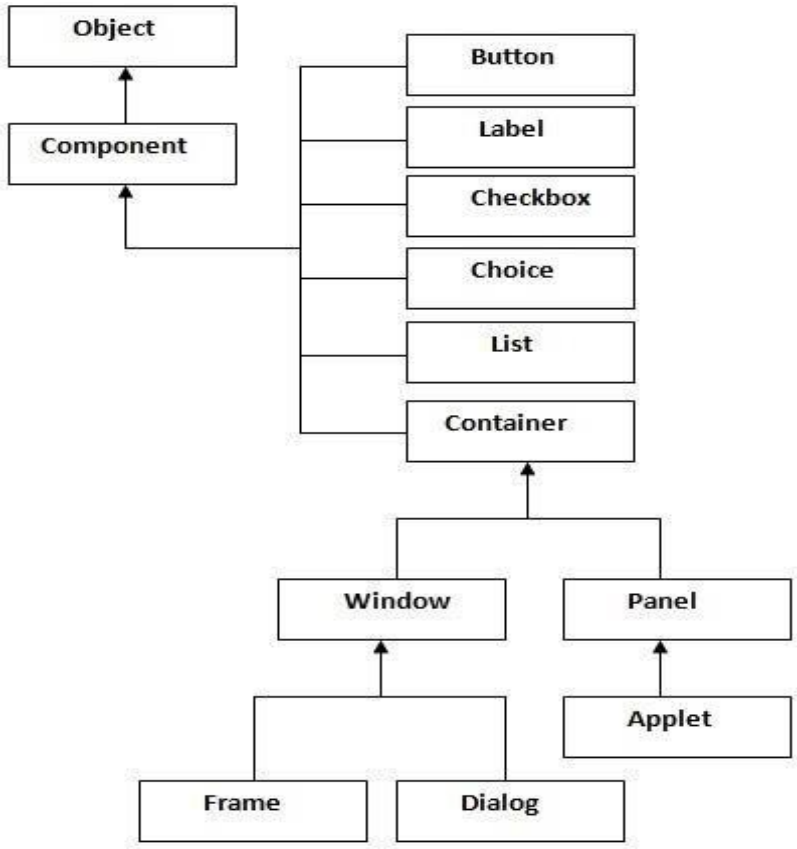
Java AWT (Abstract Window Toolkit) is *an API to develop GUI or window-based applications* in java.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.

The java.awt package provides classes for AWT api such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

Java AWT Hierarchy

The hierarchy of Java AWT classes are given below.



Container

The Container is a component in AWT that can contain another components like buttons, textfields, labels etc. The classes that extends Container class are known as container such as Frame, Dialog and Panel.

Window

The window is the container that have no borders and menu bars. You must use frame, dialog or another window for creating a window.

Panel

The Panel is the container that doesn't contain title bar and menu bars. It can have other components like button, textfield etc.

Frame

The Frame is the container that contain title bar and can have menu bars. It can have other components like button, textfield etc.

Useful Methods of Component class

Method	Description
public void add(Component c)	inserts a component on this component.
public void setSize(int width,int height)	sets the size (width and height) of the component.
public void setLayout(LayoutManager m)	defines the layout manager for the component.
public void setVisible(boolean status)	changes the visibility of the component, by default false.

Java AWT Example

To create simple awt example, you need a frame. There are two ways to create a frame in AWT.

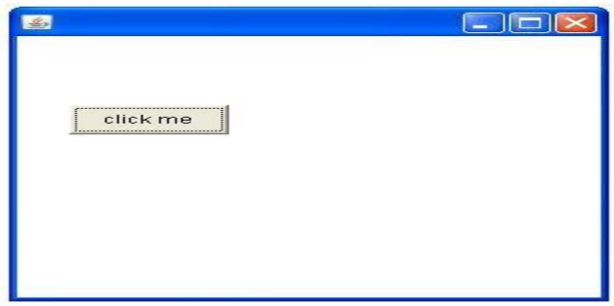
- By extending Frame class (inheritance)
- By creating the object of Frame class (association)

AWT Example by Inheritance

Let's see a simple example of AWT where we are inheriting Frame class. Here, we are showing Button component on the Frame.

```
import java.awt.*;
class First extends Frame{
First(){
Button b=new Button("click me");
b.setBounds(30,100,80,30);// setting button position
add(b);//adding button into frame
setSize(300,300);//frame size 300 width and 300 height
setLayout(null);//no layout manager
setVisible(true);//now frame will be visible, by default not visible
}
public static void main(String args[]){
First f=new First();
}}
```

The setBounds(int xaxis, int yaxis, int width, int height) method is used in the above example that sets the position of the awt button.



Java Swing

Java Swing tutorial is a part of Java Foundation Classes (JFC) that is *used to create window-based applications*. It is built on the top of AWT (Abstract Windowing Toolkit) API and entirely written in java.

Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Difference between AWT and Swing.

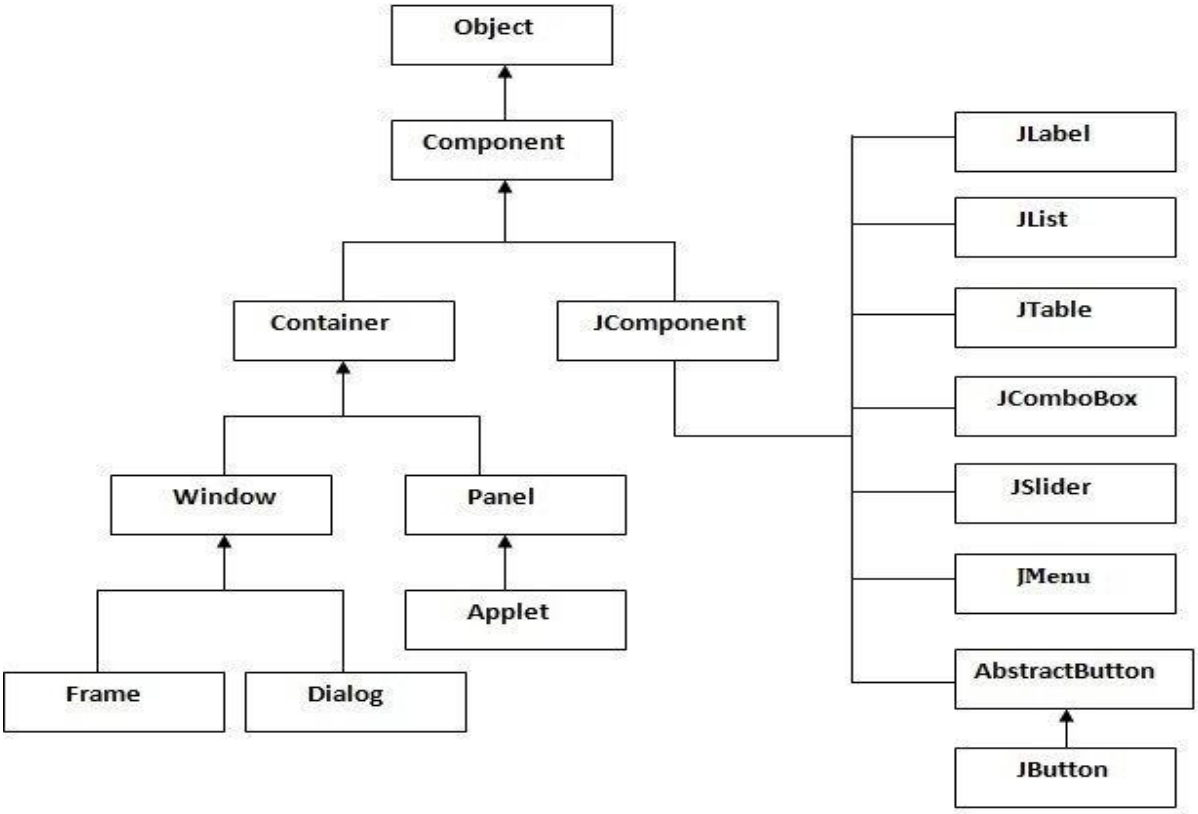
No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components**such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

Commonly used Methods of Component class

Method	Description
public void add(Component c)	add a component on another component.
public void setSize(int width,int height)	sets size of the component.
public void setLayout(LayoutManager m)	sets the layout manager for the component.
public void setVisible(boolean b)	sets the visibility of the component. It is by default false.

Hierarchy of Java Swing classes

The hierarchy of java swing API is given below.



Java Swing Examples

There are two ways to create a frame:

- By creating the object of Frame class (association)
- By extending Frame class (inheritance)

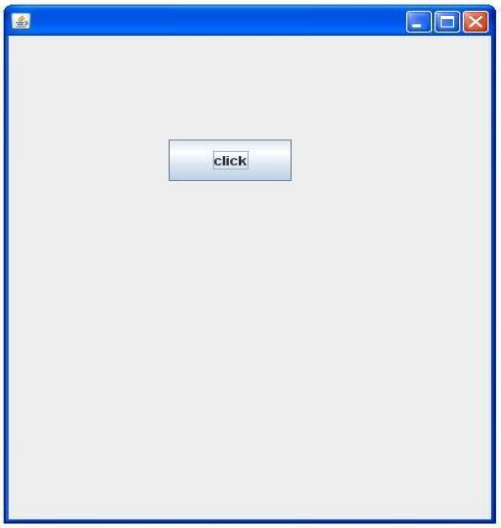
We can write the code of swing inside the main(), constructor or any other method.

Simple Java Swing Example

Let's see a simple swing example where we are creating one button and adding it on the JFrame object inside the main() method.

File: FirstSwingExample.java

```
import javax.swing.*;
public class FirstSwingExample {
public static void main(String[] args) {
JFrame f=new JFrame();//creating instance of JFrame
JButton b=new JButton("click");//creating instance of JButton
b.setBounds(130,100,100, 40);//x axis, y axis, width, height
f.add(b);//adding button in JFrame
f.setSize(400,500);//400 width and 500 height
f.setLayout(null);//using no layout managers
f.setVisible(true);//making the frame visible
} }
```



Containers

Java JFrame

The javax.swing.JFrame class is a type of container which inherits the java.awt.Frame class. JFrame works like the main window where components like labels, buttons, textfields are added to create a GUI.

Unlike Frame, JFrame has the option to hide or close the window with the help of setDefaultCloseOperation(int) method.

JFrame Example

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.Jpanel;
public class JFrameExample {
public static void main(String s[]) {
JFrame frame = new JFrame("JFrame Example");
JPanel panel = new JPanel();
panel.setLayout(new FlowLayout());
JLabel label = new JLabel("JFrame By Example");
JButton button = new JButton();
button.setText("Button");
panel.add(label);
```

```
panel.add(button);
frame.add(panel);
frame.setSize(200, 300);
frame.setLocationRelativeTo(null);
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
frame.setVisible(true);
} }
```

JApplet

As we prefer Swing to AWT. Now we can use JApplet that can have all the controls of swing. The JApplet class extends the Applet class.

Example of EventHandling in JApplet:

```
import java.applet.*;
import javax.swing.*;
import java.awt.event.*;
public class EventJApplet extends JApplet implements ActionListener{
JButton b;
JTextField tf;
public void init(){
tf=new JTextField();
tf.setBounds(30,40,150,20);
b=new JButton("Click");
b.setBounds(80,150,70,40);
add(b);add(tf);
b.addActionListener(this);
setLayout(null);
}
public void actionPerformed(ActionEvent e){
tf.setText("Welcome");
} }
```

In the above example, we have created all the controls in init() method because it is invoked only once.

myapplet.html

```
1. <html>
2. <body>
3. <applet code="EventJApplet.class" width="300" height="300">
```

</applet>
</body>
</html>

JDialog

The JDialog control represents a top level window with a border and a title used to take some form of input from the user. It inherits the Dialog class.

Unlike JFrame, it doesn't have maximize and minimize buttons.

JDialog class declaration

Let's see the declaration for javax.swing.JDialog class.

1. **public class** JDialog **extends** Dialog **implements** WindowConstants, Accessible, RootPaneConta iner

Commonly used Constructors:

Constructor	Description
JDialog()	It is used to create a modeless dialog without a title and without a specified Frame owner.
JDialog(Frame owner)	It is used to create a modeless dialog with specified Frame as its owner and an empty title.
JDialog(Frame owner, String title, boolean modal)	It is used to create a dialog with the specified title, owner Frame and modality.

Java JDialog Example

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class DialogExample {
private static JDialog d;
DialogExample() {
JFrame f= new JFrame();
d = new JDialog(f , "Dialog Example", true);
d.setLayout( new FlowLayout() );
JButton b = new JButton ("OK");
b.addActionListener ( new ActionListener()
{
public void actionPerformed( ActionEvent e )
{
                DialogExample.d.setVisible(false);
}

});

d.add( new JLabel ("Click button to continue."));
d.add(b);
d.setSize(300,300);
d.setVisible(true);
}
public static void main(String args[])
{
new DialogExample();
} }
```

Output:

JPanel

The JPanel is a simplest container class. It provides space in which an application can attach any other component. It inherits the JComponents class.

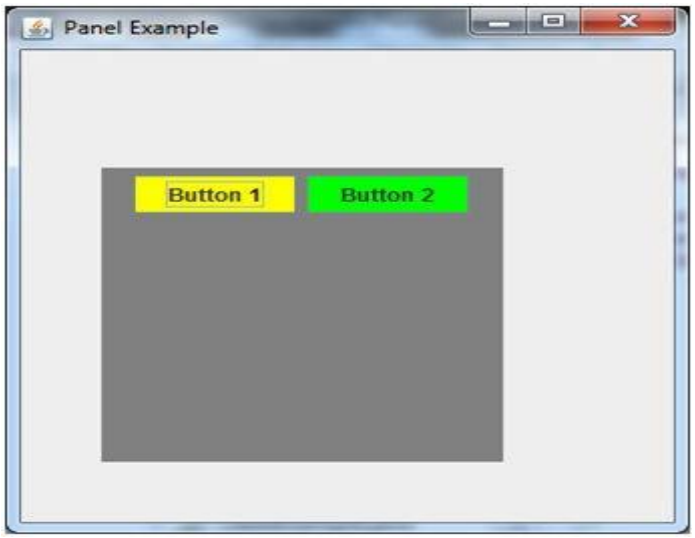
It doesn't have title bar.

JPanel class declaration

1. **public class** JPanel **extends** JComponent **implements** Accessible

Java JPanel Example

```
import java.awt.*;
import javax.swing.*;
public class PanelExample {
    PanelExample()
      {
      JFrame f= new JFrame("Panel Example");
      JPanel panel=new JPanel();
      panel.setBounds(40,80,200,200);
      panel.setBackground(Color.gray);
      JButton b1=new JButton("Button 1");
      b1.setBounds(50,100,80,30);
      b1.setBackground(Color.yellow);
      JButton b2=new JButton("Button 2");
      b2.setBounds(100,100,80,30);
      b2.setBackground(Color.green);
      panel.add(b1); panel.add(b2);
      f.add(panel);
            f.setSize(400,400);
            f.setLayout(null);
            f.setVisible(true);
      }
      public static void main(String args[])
      {
      new PanelExample();
      } }
```



Overview of some Swing Components

Java JButton

The JButton class is used to create a labeled button that has platform independent implementation. The application result in some action when the button is pushed. It inherits AbstractButton class.

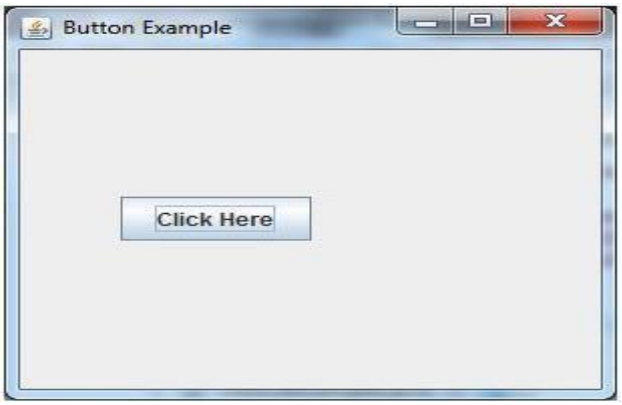
JButton class declaration

Let's see the declaration for javax.swing.JButton class.

1. **public class** JButton **extends** AbstractButton **implements** Accessible

Java JButton Example

```
import javax.swing.*;
public class ButtonExample {
public static void main(String[] args) {
    JFrame f=new JFrame("Button Example");
    JButton b=new JButton("Click Here");
    b.setBounds(50,100,95,30);
    f.add(b);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);  }  }
```



Java JLabel

The object of JLabel class is a component for placing text in a container. It is used to display a single line of read only text. The text can be changed by an application but a user cannot edit it directly. It inherits JComponent class.

JLabel class declaration

Let's see the declaration for javax.swing.JLabel class.

1. **public class** JLabel **extends** JComponent **implements** SwingConstants, Accessible

Commonly used Constructors:

Constructor	Description
JLabel()	Creates a JLabel instance with no image and with an empty string for the title.
JLabel(String s)	Creates a JLabel instance with the specified text.
JLabel(Icon i)	Creates a JLabel instance with the specified image.
JLabel(String s, Icon i, int horizontalAlignment)	Creates a JLabel instance with the specified text, image, and horizontal alignment.

Commonly used Methods:

Methods	Description
String getText()	t returns the text string that a label displays.
void setText(String text)	It defines the single line of text this component will display.
void setHorizontalAlignment(int alignment)	It sets the alignment of the label's contents along the X axis.
Icon getIcon()	It returns the graphic image that the label displays.
int getHorizontalAlignment()	It returns the alignment of the label's contents along the X axis.

Java JLabel Example

```
import javax.swing.*;
class LabelExample
{
public static void main(String args[])
   {
   JFrame f= new JFrame("Label Example");
   JLabel l1,l2;
   l1=new JLabel("First Label.");
   l1.setBounds(50,50, 100,30);
   l2=new JLabel("Second Label.");
   l2.setBounds(50,100, 100,30);
   f.add(l1); f.add(l2);
   f.setSize(300,300);
   f.setLayout(null);
   f.setVisible(true);
   }
   }
```

JTextField

The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

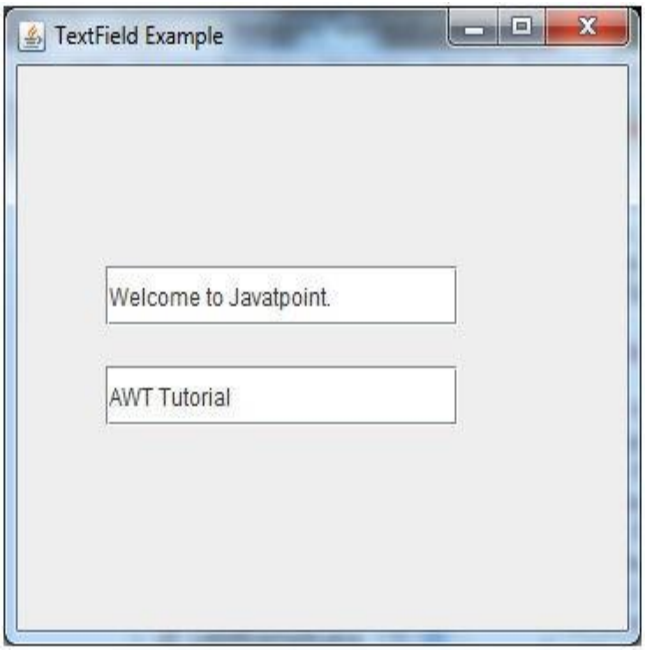
JTextField class declaration

Let's see the declaration for javax.swing.JTextField class.

1. **public class** JTextField **extends** JTextComponent **implements** SwingConstants

Java JTextField Example

```
import javax.swing.*;
class TextFieldExample
{
public static void main(String args[])
    {
    JFrame f= new JFrame("TextField Example");
    JTextField t1,t2;
    t1=new JTextField("Welcome to Javatpoint.");
    t1.setBounds(50,100, 200,30);
    t2=new JTextField("AWT Tutorial");
    t2.setBounds(50,150, 200,30);
    f.add(t1); f.add(t2);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
    }    }
```



Java JTextArea

The object of a JTextArea class is a multi line region that displays text. It allows the editing of multiple line text. It inherits JTextComponent class

JTextArea class declaration

Let's see the declaration for javax.swing.JTextArea class.

1. **public class** JTextArea **extends** JTextComponent

Java JTextArea Example

```
import javax.swing.*;
public class TextAreaExample
{
    TextAreaExample(){
       JFrame f= new JFrame();
       JTextArea area=new JTextArea("Welcome to javatpoint");
       area.setBounds(10,30, 200,200);
       f.add(area);
       f.setSize(300,300);
       f.setLayout(null);
       f.setVisible(true);
    }
public static void main(String args[])
   {
  new TextAreaExample();
   }}
```

Simple Java Applications

```
import  javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class Example extends JFrame {

   public Example() {
      setTitle("Simple example");
      setSize(300, 200);
      setLocationRelativeTo(null);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
   }

   public static void main(String[] args) {
      Example ex = new Example();
      ex.setVisible(true);
   }}
```

Layout Management

Java LayoutManagers

The LayoutManagers are used to arrange components in a particular manner. LayoutManager is an interface that is implemented by all the classes of layout managers.

BorderLayout

The BorderLayout provides five constants for each region:

1. **public static final int NORTH**
2. **public static final int SOUTH**
3. **public static final int EAST**
4. **public static final int WEST**
5. **public static final int CENTER**

Constructors of BorderLayout class:

- **BorderLayout():** creates a border layout but with no gaps between the components.
- **JBorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

Example of BorderLayout class:

```
import java.awt.*;
import javax.swing.*;
public class Border
 {
JFrame f;
Border()
{
   f=new JFrame();
   JButton b1=new JButton("NORTH");;
   JButton b2=new JButton("SOUTH");;
   JButton b3=new JButton("EAST");;
   JButton b4=new JButton("WEST");;
   JButton b5=new JButton("CENTER");;
   f.add(b1,BorderLayout.NORTH);
   f.add(b2,BorderLayout.SOUTH);
   f.add(b3,BorderLayout.EAST);
   f.add(b4,BorderLayout.WEST);
   f.add(b5,BorderLayout.CENTER);
   f.setSize(300,300);
   f.setVisible(true);
}
public static void main(String[] args)
{
  new Border();
} }
```

Output:

Java GridLayout

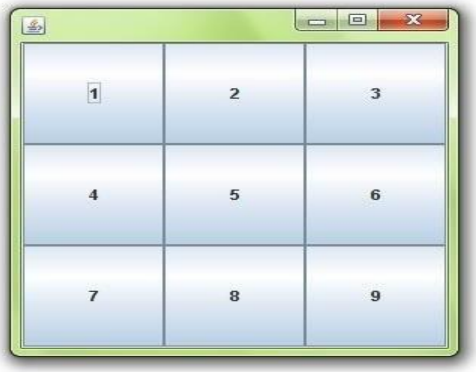
The GridLayout is used to arrange the components in rectangular grid. One component is displayed in each rectangle.

Constructors of GridLayout class

1. **GridLayout():** creates a grid layout with one column per component in a row.
2. **GridLayout(int rows, int columns):** creates a grid layout with the given rows and columns but no gaps between the components.
3. **GridLayout(int rows, int columns, int hgap, int vgap):** creates a grid layout with the given rows and columns alongwith given horizontal and vertical gaps.

Example of GridLayout class

```
1. import java.awt.*;
2. import javax.swing.*;
public class MyGridLayout{
JFrame f;
MyGridLayout(){
   f=new JFrame();
   JButton b1=new JButton("1");
   JButton b2=new JButton("2");
   JButton b3=new JButton("3");
   JButton b4=new JButton("4");
   JButton b5=new JButton("5");
   JButton b6=new JButton("6");
   JButton b7=new JButton("7");
   JButton b8=new JButton("8");
   JButton b9=new JButton("9");
   f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
   f.add(b6);f.add(b7);f.add(b8);f.add(b9);
   f.setLayout(new GridLayout(3,3));
   //setting grid layout of 3 rows and 3 columns
   f.setSize(300,300);
   f.setVisible(true);
}
public static void main(String[] args) {
 new MyGridLayout();  }}
```



Java FlowLayout

The FlowLayout is used to arrange the components in a line, one after another (in a flow). It is the default layout of applet or panel.

Fields of FlowLayout class

1. **public static final int LEFT**
2. **public static final int RIGHT**
3. **public static final int CENTER**
4. **public static final int LEADING**
5. **public static final int TRAILING**

Constructors of FlowLayout class

1. **FlowLayout():** creates a flow layout with centered alignment and a default 5 unit horizontal and vertical gap.
2. **FlowLayout(int align):** creates a flow layout with the given alignment and a default 5 unit horizontal and vertical gap.
3. **FlowLayout(int align, int hgap, int vgap):** creates a flow layout with the given alignment and the given horizontal and vertical gap.

Example of FlowLayout class

```
import java.awt.*;
import  javax.swing.*;
public class MyFlowLayout{
JFrame f;
MyFlowLayout(){
    f=new JFrame();
    JButton b1=new JButton("1");
    JButton b2=new JButton("2");
    JButton b3=new JButton("3");
    JButton b4=new JButton("4");
    JButton b5=new JButton("5");
    f.add(b1);f.add(b2);f.add(b3);f.add(b4);f.add(b5);
    f.setLayout(new FlowLayout(FlowLayout.RIGHT));
    //setting flow layout of right alignment
   f.setSize(300,300);
    f.setVisible(true);
}
public static void main(String[] args) {
new MyFlowLayout();
} }
```