Unit 1

Introduction to Java

Java is a programming language and a platform. Java is a high level, robust, object oriented and secure programming language.

Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995. James Gosling is known as the father of Java. Before Java, its name was Oak. Since Oak was already a registered company, so James Gosling and his team changed the Oak name to Java.

Platform: Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

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 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

next → ← prev

History of Java

1. History of Java

2. Java Version History

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 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.

Features of Java

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as java uzzwords.

A list of most important features of Java language is given below.

1. Simple
2. Object-Oriented
3. Portable
4. Platform independent
5. Secured
6. Robust
7. Architecture neutral
8. Interpreted
9. High Performance
10. Multithreaded
11. Distributed
12. Dynamic

Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because: Java syntax is based on C++ (so easier for programmers to learn it after C++). o Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.

There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

Object-oriented

Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination

of different types of objects that incorporates both data and behavior. Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Platform Independent

Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

The Java platform differs from most other platforms in the sense that it is a software based platform that runs on the top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code

because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).

Secured

Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

No explicit pointer, Java Programs run inside a virtual machine sandbox Robust Robust simply means strong. Java is robust because:

It uses strong memory management. There is a lack of pointers that avoids security problems.

There is automatic garbage collection in java which runs on the Java Virtual. Machine to get rid of objects which are not being used by a Java application anymore. There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

High-performance

Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

Multi-threaded

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multi-media, Web applications, etc.

Dynamic

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++. Java supports dynamic compilation and automatic memory management (garbage collection).

Java OOPs Concepts

1. Object-Oriented Programming

2. Advantage of OOPs over Procedure-oriented programming language

3. Difference between Object-oriented and Object-based programming language.

In this page, we will learn about the basics of OOPs. Object-Oriented Programming is a paradigm that provides many concepts, such as inheritance, data binding, polymorphism, etc.

OOPs (Object-Oriented Programming System)

Object means a real-world entity such as a pen, chair, table, computer, watch, etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies software development and maintenance by providing some concepts:

- Object
- Class
- Inheritance

- Polymorphism
- Abstraction
- Encapsulation

Object

Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical. An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

Example: A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

Class

Collection of objects is called class. It is a logical entity. A class can also be defined as a blueprint from which you can create an individual object. Class doesn't consume any space.

Inheritance

When one object acquires all the properties and behaviors of a parent object, it is known as inheritance.It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

If one task is performed in different ways, it is known as polymorphism. For example: to convince the customer differently, to draw something, for example, shape, triangle, rectangle, etc. In Java, we use method overloading and method overriding to achieve polymorphism. Another

example can be to speak something; for example, a cat speaks meow, dog barks woof,etc.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example phone call, we won't know the internal processing. In Java, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit are known as encapsulation. For example, a capsule, it is wrapped with different medicines. A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.
Java Environment and Java Tools

Java environments include a number of development tools, classes and methods. The development tools are part of the system known as Java Development Kit (JDK) and the classes and methods are part of the Java Standard Library(ISL), also known as the Application Programming Interface(API).

Java Development Kit The java development kit comes with a collection softools that are used for developing and running javaprogram. They include

1) Applet viewer (For viewing Java applets): A pre- created applet can be viewed using applet viewer. It enables us to run Java applets (without actually usinga Java-compatible browser). Following command is used for executing java applets:

C:\>appletviewer <URL of the.html file>

2) javac (Java Compiler): A java program can be created by using any text editor (i.e. notepad/word pad /edit pad) and save file with .java extension (filename.java). The java compiler, translates filename.java source code to

byte code files (i.e. filename. class) which theinterpreter understands. Following command is used for compiling java programs.

C:\>javac filename.java

After compilation

Filename.java----- converted to Filename.class

3) Java (Java Interpreter): The java interpreter is used to execute a java compiled application (.class) java interpreter, runs applets and applications by reading and interpreting bytecode files

Following command is used for executing javaprograms:

C:\>java filename

Note: No extension is needed while executing/runningjava programs

4) Javap (Java disassembler): Java disassembler

enables us to convert byte code file into program description

5) Javah (for c header files): It produces headerfiles for using with native methods.

6) Javadoc (for creating HTML documents): Itcreates HTML format documentation from java source code files

7) Jdb (Java debugger): Java debugger, helps us tofind errors in our programs.

First Java Program | Hello World Example The requirement for Java Hello World Example For executing any java program, you need to Install the JDK if you don't have installed it, download the JDK and install it.

- Set path of the jdk/bin directory
- Create the java program
- Compile and run the java program

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

Steps:

Save this file as Simple.java

To compile: javac Simple.java

To execute: java Simple

Compilation Flow:

When we compile Java program using javac tool, java compiler converts the source code into byte code.

Parameters used in First Java Program

Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

import keyword is used to import a package, class or interface.

import java package into a class, we need to use java import keyword which is used to access package and its classes into the java program.

Use import to access built-in and user-defined packages into your java source file

- class keyword is used to declare a class in java.
- public keyword is an access modifier which represents visibility. It means it is visible to all.
- static is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method.The main method is executed by the JVM, so it doesn't require to create an object to invoke the main method. So it saves memory.
- void is the return type of the method. It means it doesn't return any value.
- main represents the starting point of the program.
- String[] args is used for command line argument.
- System.out.println() is used to print statement. Here, System is a class, out is the object of printStream class, println() is the method of PrintStream class.

Difference between JDK, JRE, and JVM

JVM:

JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java byte code can be executed. The JVM performs the following main tasks:

- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JRE

JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which

are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a

set of libraries +other files that JVM uses at runtime. The implementation of JVM is also actively released by other companies besides Sun Micro Systems.

JDK

JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:

- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader(java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application. Application Programming Interface (API).

The java Standard Library (JSL) or API includes hundreds of classes and methods several unctional packages. Most commonly used packages are:

i. Language support package: A collection of classes and methods required for implementing basic features of Java.

2) Utilities package: A collection of classes to provide utility functions such as date and time

3) Input /output package: A collection of classes required for input/output manipulation.

4) Networking package: A collection of classes for communicating with other compilers via Internet

5) AWT package: The Abstract Window Toolkit (AWT) package contains classes that implements platform-independent graphical user interface.

6) Applet package: This includes a set of classes that allows us to create java applets. Java Support Systems. The operation of java and java enabled browsers on the internet requires a variety of support system Following lists the systems necessary to support java for delivering information on the internet

i. Internet connection: Local computer should be connected to the internet

ii. Web server: A program that accepts requests for information and sends the required documents.

iii. Web browser: A program that provides access to WWW and runs java applets.

iv. HTML: A language for creating hypertext for the web.

v. Applet tag: It is used for placing java applets in HTML documents.

vi. Java code: Java code is used for defining java applets.

vii. Byte code: Compiled java code that is referred to in the APPLET tag and transferred to the user computer.

Variables

A variable is a container which holds the value while the Java program is executed. A variable is assigned with a data type. Variable is a name of memory location. There are three types of variables in java: local, instance

and static. There are two types of data types in Java: primitive and non-primitive.

Variable

Variable is name of reserved area allocated in memory. In other words, it is a name of memory location. It is a combination of "vary + able" that means its value can be changed.

1. int data=50;//Here data is variable Types of Variables

There are three types of variables in Java:

- local variable
- instance variable
- static variable

1) Local Variable

- A variable declared inside the body of the method is called local variable.
- Local variable can be declared in the method, Constructor or block Local variables are visible only within the declared method constructor or block
- You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.
- A local variable cannot be defined with "static" keyword.
- Access modifiers cannot be used with local variable

Let us take an example in which we take local variable to explain it. Here, age is a local variable. This variable is defined under putAge() method and its scope is limited to this method only:

```
public class Dog
{
public void putAge()
{
int age = 0; //local variable
age = age + 6;
System.out.println("Dog age is : " + age);
}
public static void main(String args[]){Dog d = new Dog();
d.putAge();
}
}
```

Output:

Dog age is :6

2) Instance Variable

- A variable declared inside the class but outside the body of the method, is called instance variable.
- It is not declared as static.
- It is called instance variable because its value is instance specific and is not shared among instances.
- Instance variable can be accessed object with reference variable which contains object

```
import java.io.*;
public class Employee {
// this instance variable is visible for any child class.public
String name;
//salary variable is visible in Employee class only.private
double salary;
// The name variable is assigned in the constructor.public
Employee (String empName)
{
name = empName;
}
// The salary variable is assigned a value.public
void setSalary(double empSal)
{
salary = empSal;
}
// This method prints the employee details.public void
printEmp()
{
System.out.println("name : " + name );System.out.println("salary :"
+ salary);
}
```

```
public static void main(String args[])
{
Employee empOne = new Employee("Ransika");
empOne.setSalary(1000);
empOne.printEmp();
}
}
```

Output

This will produce the following result –name :

Ransika

salary :1000.0

3) Static variable

- A variable which is declared as static is called static variable.
- A variable declared inside the class with static keyword are called as static variable
- It cannot be local.
- You can create a single copy of static variable and share among all the instances of the class.
- Memory allocation for static variable happens only once when the class is loaded in the memory.
- Static variables can be accessed with following three ways With class name 2with reference variable which contain null

3) with reference variable which contains object Static variables can be accessed by calling with the class name

Classname.variablename

```
import java.io.*; public
class Employee
{
//salary variable is a private static variableprivate
static double salary;
// DEPARTMENT is a constant
public static final String DEPARTMENT = "Development ";public
static void main(String args[]) {
salary = 1000;
System.out.println(DEPARTMENT + "average salary:" + salary);
}
}
```

Output

This will produce the following result −

Development average salary:1000

Note − If the variables are accessed from an outside class, the constant should be accessed as Employee.DEPARTMENT

Example to understand the types of variables in java

```
class A{
int data=50;//instance variable
static int m=100;//static variable
```

```
 void method(){
int n=90;//local variable6.
}
}
//end of class
```

Java Variable Example: Add Two Numbers

```
class Simple{
 public static void main(String[] args){
int a=10;
int b=10;
int c=a+b;
System.out.println(c); 7.
}
 }
```

Output: 20

Data Types

Data Type determines type of value that a variable can hold The main purpose of data types in java is to determine what kind of value we can stored in to the variable. Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. Primitive data types:

Primitive data types are predefined by the language and named by keyword primitive data types and are used to store actual value. These types are also called as built in data type. The primitive data types include boolean, char, byte, short, int, long, float and double.

2. Non-primitive data types:

These data types are also called as derived data types. These are called reference data type because they contain address of value rather than value itself. Reference data types are made using primitive data types

The non-primitive data types include Classes, Interfaces,and

Arrays,Strings

There are 8 types of primitive data types:

- boolean data type
- byte data type
- char data type
- short data type
- int data type
- long data type
- float data type
- double data type

Java Control Statements | Control Flow in Java

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements. It is one of the fundamental features of Java, which provides a smooth flow of program.

Java provides three types of control flow statements.

1. Decision Making statements
   - if statements
   - switch statement
2. Loop statements
   - do while loop
   - while loop
   - for loop
   - for-each loop
3. Jump statements
   - break statement
   - continue statement

Decision-Making statements:

As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) If Statement:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement

3. if-else-if ladder
4. Nested if-statement

Let's understand the if-statements one by one.

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
if(condition)
 {
statement 1; //executes when condition is true
}
```

Consider the following example in which we have used the **if** statement in the java code.

Student.java

Student.java

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y > 20) {
System.out.println("x + y is greater than 20");
}
```

```
}
}
```

Output:

```
x + y is greater than 20
```

2) if-else statement

The if-else statement

is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition)
 {
statement 1; //executes when condition is true
}
Else
{
statement 2; //executes when condition is false
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y < 10) {
System.out.println("x + y is less than      10");
}   else {
System.out.println("x + y is greater than 20");
}
}
    }
```

Output:

```
x + y is greater than 20
```

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax of if-else-if statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
```

```
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
}
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
String city = "Delhi";
if(city == "Meerut") {
System.out.println("city is meerut");
}else if (city == "Noida") {
System.out.println("city is noida");
}else if(city == "Agra") {
System.out.println("city is agra");
}else {
System.out.println(city);
}
    }
}
```

Output:

```
Delhi
```

4. Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax of Nested if-statement is given below.

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true
}
else{
statement 2; //executes when condition 2 is false
}  }
```

Consider the following example.

Student.java

```
public class Student {
public static void main(String[] args) {
String address = "Delhi, India";

if(address.endsWith("India")) {
if(address.contains("Meerut")) {
System.out.println("Your city is Meerut");
}else if(address.contains("Noida")) {
System.out.println("Your city is Noida");
}else {
```

```
System.out.println(address.split(",")[0]);
}
}else {
System.out.println("You are not living in India");
}  }  }
```

Output:

```
Delhi
```

Switch Statement:

In Java, Switch statements

are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied.
  It is optional, if not used, next case is executed.
- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

The syntax to use the switch statement is given below.

```
switch (expression){
   case value1:
    statement1;
    break;
   .
  .
   .
   case valueN:
    statementN;
    break;
   default:
    default statement;
}
```

Consider the following example to understand the flow of the switch statement.

Student.java

```
public class Student implements Cloneable {
public static void main(String[] args) {
int num = 2;
switch (num){
case 0:
System.out.println("number is 0");
break;
case 1:
System.out.println("number is 1");
break;
```

default:
System.out.println(num);
}
}
}

Output:

2

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

1. for loop
2. while loop
3. do-while loop

Let's understand the loop statements one by one.

Java for loop

In Java, for loop. It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```

Calculation.java

```
public class Calculattion
 {
public static void main(String[] args)
 {
// TODO Auto-generated method stub
int sum = 0;
    for(int j = 1; j<=10; j++)
     {
    sum = sum + j;
    }
    System.out.println("The sum of first 10 natural numbers is " + sum);
}
}
```

Output:

```
The sum of first 10 natural numbers is 55
```

Java Single Line Comment

The single-line comment is used to comment only one line of the code. It is the widely used and easiest way of commenting the statements.

Single line comments starts with two forward slashes **(//)**. Any text in front of // is not executed by Java.

Syntax:

```
//This is single line comment
```

Let's use single line comment in a Java program.

CommentExample1.java

```
public class CommentExample1
 {
public static void main(String[] args)
 {
    int i=10; // i is a variable with value 10
System.out.println(i);  //printing the variable i
}
}
```

Output:

```
10
```

Multi Line Comment

The multi-line comment is used to comment multiple lines of code. It can be used to explain a complex code snippet or to comment multiple lines of code at a time (as it will be difficult to use single-line comments there).

Multi-line comments are placed between /* and */. Any text between /* and */ is not executed by Java.

Syntax:

```
/*
This
is
multi line   comment  */
```

UNIT 2

Classes and Objects

An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.

An object has three characteristics:

- **State:** represents the data (value) of an object.
- **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
- **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

Object Definitions:

- An object is *a real-world entity*.
- An object is *a runtime entity*.
- The object is *an entity which has state and behavior*.
- The object is *an instance of a class*.

class in Java

A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

Syntax to declare a class:

```
class <class_name>
{
   field;
   method;
}
```

Object and Class Example: main within the class

In this example, we have created a Student class which has two data members id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

File: Student.java

```
//Java Program to illustrate how to define a class and fields
//Defining a Student class.
```

```
class Student
{
 //defining fields
     int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[])
{
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
       System.out.println(s1.name);
 }
}
```

OUTPUT

0

Null

Constructors in Java

In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

It is a special type of method which is used to initialize the object.

Every time an object is created using the new() keyword, at least one constructor is called.

It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

There are two types of constructors in Java: no-arg constructor, and parameterized constructor.

Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (no-arg constructor)
2. Parameterized constructor

Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Syntax of default constructor:

<class_name>(){}

Example of default constructor

In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation

```
//Java Program to create and call a default constructor
class Bike1
{
//creating a default constructor
Bike1(){System.out.println("Bike is created");
}
//main method
public static void main(String args[])
{
//calling a default constructor
Bike1 b=new Bike1();
}
}
```

```
1. Output:
2. Bike is created
```

Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor.

Why use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example of parameterized constructor

In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.

//Java Program to demonstrate the use of the parameterized constructor.

```
class Student4
{
   int id;
   String name;
   //creating a parameterized constructor
   Student4(int i,String n)
{
   id = i;
   name = n;
   }
   //method to display the values
   void display(){System.out.println(id+" "+name);
}

   public static void main(String args[])
{
   //creating objects and passing values
   Student4 s1 = new Student4(111,"Karan");
   Student4 s2 = new Student4(222,"Aryan");
   //calling method to display the values of object
   s1.display();
   s2.display();
  }
}
```

Output:

1111 Karan
222 Aryan

Inheritance in Java

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

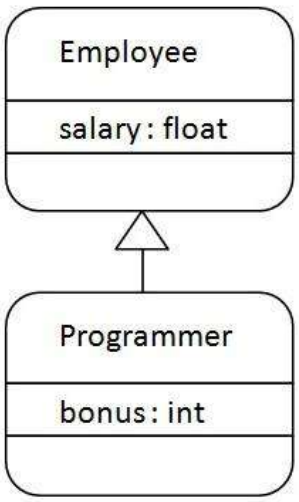
The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

Java Inheritance Example



11 Karan

222 Aryan

As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
class Employee
{
 float salary=40000;
}
class Programmer extends Employee
{
 int bonus=10000;
 public static void main(String args[])
{
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

OutPut:

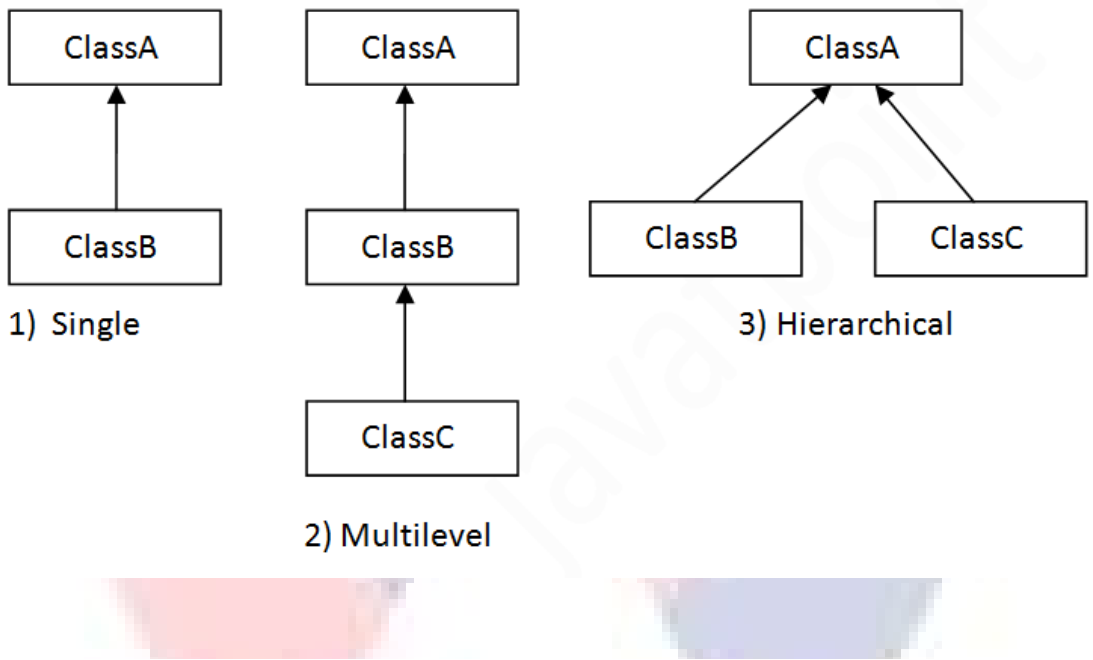
y is:40000.0

Bonus of programmer is:10000

Types of inheritance in java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.



Single Inheritance

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

Single inheritance is the simplest type of inheritance in java. In this, a class inherits the properties from a single class. The class which inherits is called the derived class or child class or subclass, while the class from which the derived class inherits is called the base class or superclass or parent class. So, in single inheritance, we have only one derived class and one base class.

Syntax of Single Inheritance in Java

```
class base class
{
    .... methods
```

```
}

class derivedClass name extends baseClass

{

   methods ... along with this additional feature

}
```

Java uses the keyword "**extends**" to create a new class(derived class) from the existing class(base class). The term "**extends**" means to increase the functionality as the derived class can reuse the methods and fields of the base class, and along with this, new methods and fields can also be defined in the derived class, hence increasing the functionality.

Code to illustrate Single Inheritance in Java

```
 class Employee {

   void salary() {

      System.out.println("Salary= 200000");

   }

}


class Programmer extends Employee {

// Programmer class inherits from Employee class

   void bonus() {

      System.out.println("Bonus=50000");
```

```
    }
}

class single_inheritance {
    public static void main(String args[]) {
        Programmer p = new Programmer();
        p.salary(); // calls method of super class
        p.bonus(); // calls method of sub class
    }
}
```

Output:
Salary= 200000
Bonus=50000
Multilevel Inheritance

Multilevel Inheritance in java involves inheriting a class, which already inherited some other class. Correlating it with a real-life scenario, we've often seen some of our habits and thoughts match precisely with our parents. And similarly, their habits match with their parents, i.e., our grandparents.
We can create hierarchies in Java with as many layers of Inheritance as we want. This means we can utilize a subclass as a superclass. In this case, each subclass inherits all of the traits shared by all of its super classes.

.

There's a family hierarchy:
Sachin → Raj(Sachin's Father) → Shyam(Raj's Father) →
Ramesh(Shyam's Father).
Here, Sachin is the child of Raj.
Raj is the child of Shyam.
Shyam is the child of Ramesh.
For consideration in this example, Ramesh has his own traits.
Shyam has his own traits, along with all the traits of Ramesh.
Raj has his own traits, along with all the traits of Shyam.
Sachin has his own traits, along with all the traits of Raj.
This is one example of multilevel inheritance that we can visualize in our day-to-day life.

```
//Superclass box...

class  Box {

   private  double width;

   private  double height;

   private  double depth;

   // Constructor of the superclass

   Box(double w,  double h,  double d)

   {

      width = w;

      height = h;
```

```
        depth = d;
    }
    // Volume calculation...
    double  volume() {
        return width * height * depth;
    }
}
// Sub class - 1
// BoxWeight class extending the box class...
class  BoxWeight  extends  Box
{
    double weight;  // weight of box
    // Sub class -1 constructor
    BoxWeight(double w,  double h,  double d,  double m) {
        super(w, h, d);  // calling superclass constructor
        weight = m;
    }
}
// Sub class - 2
```

```
// Shipment class extending BoxWeight class...
class  Shipment  extends  BoxWeight {
  double cost;
  // Sub class - 2 constructor
  Shipment(double w,  double h,  double d,  double m,  double c) {
     super(w, h, d, m);  // calling superclass constructor
     cost = c;
  }}

public class  TestMultilevel
{
  public  static  void  main(String args[])
  {
     Shipment shipment1 =  new Shipment(1,  2,  3,  5,  3.41);
     Shipment shipment2 =  new Shipment(2,  4,  6,  10,  1.28);
     double vol;
     vol = shipment1.volume();
     System.out.println("The volume of shipment1 is " + vol);
```

```
        System.out.println("The weight of shipment1 is " +
shipment1.weight);

        System.out.println("Shipping cost: Rs." + shipment1.cost);

        System.out.println();

        vol = shipment2.volume();

        System.out.println("The volume of shipment2 is " + vol);

        System.out.println("The weight of shipment2 is " +
shipment2.weight);

        System.out.println("Shipping cost: Rs." + shipment2.cost);

    }

}
```

Output

```
The volume of shipment1 is 6.0

The weight of shipment1 is 5.0

Shipping cost: Rs.3.41

The volume of shipment2 is 48.0

The weight of shipment2 is 10.0

Shipping cost: Rs.1.28
```

Why Java Does Not Support Multiple Inheritance?
The main reason for not allowing multiple inheritances in java is the ambiguity around the **diamond problem.**
Consider a class NinjaA that contains a Coding() method. Classes NinjaB and NinjaC were derived from class NinjaA and each had a Coding() implementation. Class NinjaD is now derived from classes NinjaB and NinjaC using multiple inheritances. If we refer to simply Coding(), the compiler would not be able to determine which Coding() it should call. Because of the inheritance scenario's structure, which resembles a four-edged diamond, this problem is also known as the "Diamond Problem."

Interface

An Interface *is the blueprint of a class that implements* abstraction *by using abstract methods. Declaring interfaces is easy and can be done by simply using the 'interface' keyword.*

Now you may be wondering how using an interface solves the problem we were facing before.
To understand the reason, let us consider our previous example, but we'll use an interface instead of a class this time.

```
interface A   //First interface
{
   default void text()
   {
            System.out.println("Hello");
   }
```

```
}

interface B   //Second interface
{
   default void text()
   {
            System.out.println("What's your name?");
   }
}

class C implements A,B
{

   public void text()   //Default method in interface is overriden
   {
            A.super.text();  //text() method from first interface is called
            B.super.text();  //text() method from second interface is called
   }
}
```

```
class Main
{
   public static void main(String args[])
        {
        C obj = new C();
        obj.text();
   }
}
```

Output:

Hello

What's your name?

Abstract Class and Abstract Methods

Java Abstract Class

The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the abstract keyword to declare an abstract class. For example,

// create an abstract class

```
abstract class Language {
  // fields and methods
}
...

// try to create an object Language
// throws an error
Language obj = new Language();
```

An abstract class can have both the regular methods and abstract methods. For example,

```
abstract class Language {

  // abstract method
  abstract void method1();

  // regular method
  void method2() {
    System.out.println("This is regular method");
  }
```

}

To know about the non-abstract methods, visit Java methods. Here, we will learn about abstract methods.

Java Abstract Method

A method that doesn't have its body is known as an abstract method. We use the same abstract keyword to create abstract methods. For example,

abstract void display();

Here, display() is an abstract method. The body of display() is replaced by ;.

If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. For example,

// error

// class should be abstract

class Language {

// abstract method

abstract void method1();

}

Example: Java Abstract Class and Method

Though abstract classes cannot be instantiated, we can create subclasses from it. We can then access members of the abstract class using the object of the subclass. For example,

```
abstract class Language {


  // method of abstract class
  public void display() {
    System.out.println("This is Java Programming");
  }
}


class Main extends Language {
  public static void main(String[] args)
{
        // create an object of Main
    Main obj = new Main();


    // access method of abstract class
```

```
    // using object of Main class
    obj.display();
  }
}
```

Output

This is Java programming

Method overloading allows the method to have the same name which differs on the basis of arguments or the argument types. It can be related to compile-time polymorphism. Following are a few pointers that we have to keep in mind while overloading methods in Java.

- We cannot overload a return type.
- Although we can overload static methods, the arguments or input parameters have to be different.
- We cannot overload two methods if they only differ by a static keyword.
- Like other static methods, the main() method can also be overloaded.

```
public class Div{
public int div(int a , int b){
        return (a / b); }
 public int div(int a , int b , int c){
        return ((a + b ) / c); }
```

```
public static void main(String args[]){
Div ob = new Div();
ob.div(10 , 2);
ob.div(10, 2 , 3);
}
}
```

Output: 5

4

The main advantage of using method overloading in Java is that it gives us the liberty to not define a function again and again for doing the same thing. In the below example, the two methods are basically performing division, so we can have different methods with the same name but with different parameters. It also helps in compile-time polymorphism.

```
public class Test{
public static int func(int a ){
    return 100;
}
public static char func(int a , int b){
    return "edureka";
```

```
}
public static void main(String args[]){
System.out.println(func(1));
System.out.println(func(1,3));
}
}
```

Output: 100

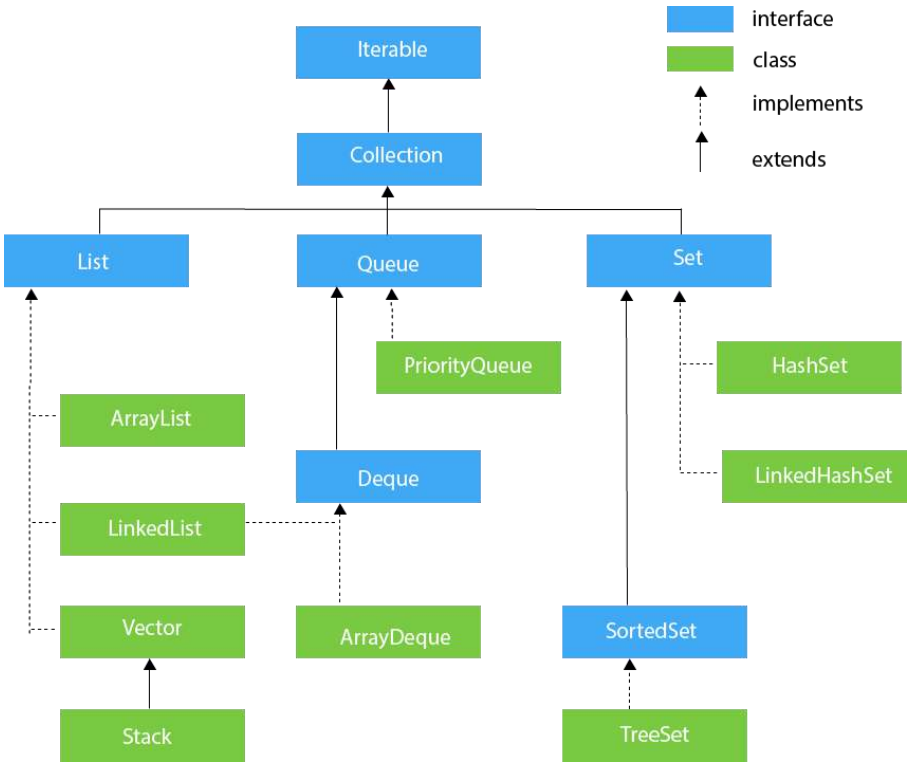
Edureka

UNIT 3

Collection

Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

Java Collection means a single unit of objects. Java Collection framework provides many interfaces (Set, List, Queue, Deque) and classes (ArrayList, Vector, LinkedList, PriorityQueue, HashSet, LinkedHashSet, TreeSet).



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

Vector

Vector uses a dynamic array to store the data elements. It is similar to ArrayList. However, It is synchronized and contains many methods that are not the part of Collection framework.

Consider the following example.

```
import java.util.*;
public class TestJavaCollection3
{
public static void main(String args[])
{
```

```
        Vector<String> v=new Vector<String>();
        v.add("Ayush");
        v.add("Amit");
        v.add("Ashish");
        v.add("Garima");
        Iterator<String> itr=v.iterator();
    while(itr.hasNext()){
    System.out.println(itr.next());
    }
    }
        }
```

Output:

Ayush

Amit

Ashish

Garima

HashSet

Learn more

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

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

Hashtable class

Java Hashtable class implements a hashtable, which maps keys to values. It inherits Dictionary class and implements the Map interface.

- A Hashtable is an array of a list. Each list is known as a bucket. The position of the bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
- Java Hashtable class contains unique elements.

- Java Hashtable class doesn't allow null key or value.
- Java Hashtable class is synchronized.
- The initial default capacity of Hashtable class is 11 whereas loadFactor is 0.75.

Hashtable class declaration

Let's see the declaration for java.util.Hashtable class.

1. **public class** Hashtable<K,V> **extends** Dictionary<K,V> **implements** Map <K,V>, Cloneable, Serializable

Hashtable class Parameters

Let's see the Parameters for java.util.Hashtable class.

K: It is the type of keys maintained by this map.

V: It is the type of mapped values.

```
import java.util.*;
class Hashtable1{
 public static void main(String args[]){
 Hashtable<Integer,String> hm=new Hashtable<Integer,String>();
 hm.put(100,"Amit");
 hm.put(102,"Ravi");
 hm.put(101,"Vijay");
hm.put(103,"Rahul");
  for(Map.Entry m:hm.entrySet())
{
```

```
        System.out.println(m.getKey()+" "+m.getValue());
      }
    }
}
```

Output:

103 Rahul

102 Ravi

101 Vijay

100 Amit

Map Interface

A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.A Map is useful if you have to search, update or delete elements on the basis of a key.

Java **HashMap** class implements the Map interface which allows us *to store key and value pair*, where keys should be unique. If you try to insert the duplicate key, it will replace the element of the corresponding key. It is easy to perform operations using the key index like updation, deletion, etc. HashMap class is found in the java.util package.

HashMap in Java is like the legacy Hashtable class, but it is not synchronized. It allows us to store the null elements as well, but there should be only one null key. Since Java 5, it is denoted as HashMap<K,V>, where K stands for key and V for value. It inherits the AbstractMap class and implements the Map interface.

- Java HashMap contains values based on the key.
- Java HashMap contains only unique keys.
- Java HashMap may have one null key and multiple null values.
- Java HashMap is non synchronized.
- Java HashMap maintains no order.
- The initial default capacity of Java HashMap class is 16 with a load factor of 0.75.

```
import java.util.*;

public class HashMapExample1{

 public static void main(String args[]){

   HashMap<Integer,String>                                    map=new
HashMap<Integer,String>();//Creating HashMap

   map.put(1,"Mango");  //Put elements in Map

   map.put(2,"Apple");

   map.put(3,"Banana");

   map.put(4,"Grapes");


   System.out.println("Iterating Hashmap...");

   for(Map.Entry m : map.entrySet()){

    System.out.println(m.getKey()+" "+m.getValue());
```

```
    }
}
}
```

OutPut:

```
Iterating Hashmap...
1 Mango
2 Apple
3 Banana
4 Grapes
```

UNIT 4

File and Exception Handling

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that the normal flow of the application can be maintained.In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

Dictionary Meaning: Exception is an abnormal condition.In Java, an exception is an event that disrupts the normal flow of the program. It is an object which is thrown at runtime.

Exception Handling is a mechanism to handle runtime errors such as

ClassNotFoundException, IOException, SQLException, RemoteException, etc.

Advantage of Exception Handling

The core advantage of exception handling is to maintain the normal flow of the application. An exception normally disrupts the normal flow of the application; that is why we need to handle exceptions. Let's consider a scenario:

statement 1;

statement 2;

statement 3;

statement 4;

statement 5;//exception occurs

statement 6;

statement 7;

statement 8;

statement 9;

statement 10;

Suppose there are 10 statements in a Java program and an exception occurs at statement 5; the rest of the code will not be executed, i.e., statements 6 to 10 will not be executed. However, when we perform exception handling, the rest of the statements will be executed. That is why we use exception handling in Java.

Hierarchy of Java Exception classes

The java.lang.Throwable class is the root class of Java Exception hierarchy inherited by two subclasses: Exception and Error. The hierarchy of Java Exception classes is given below:

Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. An error is considered as the unchecked exception. However, according to Oracle, there are three types of exceptions namely:

1. Checked Exception

2. Unchecked Exception

3. Error

Difference between Checked and Unchecked Exceptions

1) Checked Exception

The classes that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions.

For example, ArithmeticException, NullPointerException,

ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

Java Exception Keywords

Java provides five keywords that are used to handle the exception. The following table describes each.

Keyword Description

try The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.

catch The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.

finally The "finally" block is used to execute the necessary code of the program.

It is executed whether an exception is handled or not.

throw The "throw" keyword is used to throw an exception.

throws The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

JavaExceptionExample.java

```
public class JavaExceptionExample
{
public static void main(String args[])
{
try
{
//code that may raise exception
```

```
        int data=100/0;
        }
        catch(ArithmeticException e)
        {
        System.out.println(e);
        }
        //rest code of the program
        System.out.println("rest of the code...");
        }
}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero

rest of the code...

In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
1. int a=50/0;//ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;

System.out.println(s.length());//NullPointerException
```

3) A scenario where NumberFormatException occurs

If the formatting of any variable or number is mismatched, it may result into NumberFormatException. Suppose we have a string variable that has characters; converting this variable into digit will cause NumberFormatException.

```
1. String s="abc";

2. int i=Integer.parseInt(s);//NumberFormatException
```

4) A scenario where ArrayIndexOutOfBoundsException occurs

When an array exceeds to it's size, the ArrayIndexOutOfBoundsException occurs. there may be other reasons to occur ArrayIndexOutOfBoundsException. Consider the following statements.

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

a[10]=50; //ArrayIndexOutOfBoundsException

Java try-catch block

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

If an exception occurs at the particular statement in the try block, the rest of the block code will not execute. So, it is recommended not to keep the code in try block that will not throw an exception.

Java try block must be followed by either catch or finally block.

Syntax of Java try-catch

```
Try
{
//code that may throw an exception
}
catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
Try
{
//code that may throw an exception
}
```

finally{}

Java catch block

Java catch block is used to handle the Exception by declaring the type of exception within the parameter. The declared exception must be the parent class exception (i.e., Exception) or the generated exception type. However, the good approach is to declare the generated type of exception.

The catch block must be used after the try block only. You can use multiple catch block with a single try block.

Internal Working of Java try-catch block

The JVM firstly checks whether the exception is handled or not. If exception is not handled, JVM provides a default exception handler that performs the following tasks:

Prints out exception description.

Prints the stack trace (Hierarchy of methods where the exception occurred).

Causes the program to terminate.

But if the application programmer handles the exception, the normal flow of the application is maintained, i.e., rest of the code is executed.

Problem without exception handling

Let's try to understand the problem if we don't use a try-catch block.

Example 1

TryCatchExample1.java

```
public class TryCatchExample1
{
public static void main(String[] args)
{
int data=50/0; //may throw exception
System.out.println("rest of the code");
}
}
```

Output:

Exception in thread "main" java.lang.ArithmeticException: / by zero

As displayed in the above example, the rest of the code is not executed (in such case, the rest of the code statement is not printed).

There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed. Solution by exception handling

Let's see the solution of the above problem by a java try-catch block.

Example 2

TryCatchExample2.java

```
public class TryCatchExample2
{
public static void main(String[] args)
{
try
{
int data=50/0; //may throw exception
}
//handling the exception
catch(ArithmeticException e)
{
System.out.println(e);
}
System.out.println("rest of the code");
}
}
```

Output:

```
java.lang.ArithmeticException: / by zero
rest of the code
```

As displayed in the above example, the rest of the code is executed, i.e., the rest of the code statement is printed.

Example 3

In this example, we also kept the code in a try block that will not throw an exception.

TryCatchExample3.java

```
public class TryCatchExample3
 {
public static void main(String[] args)
 {
try
{
int data=50/0; //may throw exception
// if exception occurs, the remaining statement will not execute
System.out.println("rest of the code");
}
// handling the exception
catch(ArithmeticException e)
{
```

```
System.out.println(e);
}
}
}
```

Output:

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

Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

At a time only one exception occurs and at a time only one catch block is executed.

All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

Flowchart of Multi-catch Block

Example 1

Let's see a simple example of java multi-catch block.

MultipleCatchBlock1.java

```
public class MultipleCatchBlock1
```

```
{
public static void main(String[] args)
{
Try
{
int a[]=new int[5];
a[5]=30/0;
}
catch(ArithmeticException e)
{
System.out.println("Arithmetic Exception occurs");
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("ArrayIndexOutOfBounds Exception occurs");
}
catch(Exception e)
{
System.out.println("Parent Exception occurs");
```

```
}
System.out.println("rest of the code");
}
}
```

Output:

10 Sec

23.2M

443

Difference between JDK, JRE, and JVM

Arithmetic Exception occurs

rest of the code

Example 2

MultipleCatchBlock2.java

```
 public class MultipleCatchBlock2
 {
public static void main(String[] args)
 {
try{
int a[]=new int[5];
```

```
System.out.println(a[10]);
}
catch(ArithmeticException e)
{
System.out.println("Arithmetic Exception occurs");
}
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("ArrayIndexOutOfBounds Exception occurs");
}
catch(Exception e)
{
System.out.println("Parent Exception occurs");
}
System.out.println("rest of the code");
}
}
```

Output:

ArrayIndexOutOfBounds Exception occurs rest of the code

Java finally block

Java finally block is a block used to execute important code such as closing the connection, etc.

Java finally block is always executed whether an exception is handled or not.

Therefore, it contains all the necessary statements that need to be printed regardless of the exception occurs or not.

The finally block follows the try-catch block.

Flowchart of finally block

Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Why use Java finally block?

finally block in Java can be used to put "cleanup" code such as closing a file, closing connection, etc.

The important statements to be printed can be placed in the finally block.

Usage of Java finally

Let's see the different cases where Java finally block can be used.

TestFinallyBlock.java

```
class TestFinallyBlock
{
```

```
public static void main(String args[])
{
Try
{
//below code do not throw any exception
int data=25/5;
System.out.println(data);
}
//catch won't be executed
catch(NullPointerException e)
{
System.out.println(e);
}
//executed regardless of exception occurred or not
finally
{
System.out.println("finally block is always executed");
}
System.out.println("rest of phe code...");
```

```
 }

}
```

Output:

Java throw Exception

In Java, exceptions allows us to write good quality codes where the errors are checked at the compile time instead of runtime and we can create custom exceptions making the code recovery and debugging easier.

Java throw keyword

The Java throw keyword is used to throw an exception explicitly.

We specify the exception object which is to be thrown. The Exception has some message with it that provides the error description. These exceptions may be related to user inputs, server, etc.

We can throw either checked or unchecked exceptions in Java by throw keyword. It is mainly used to throw a custom exception. We will discuss custom exceptions later in this section. We can also define our own set of conditions and throw an exception explicitly using throw keyword. For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.

The syntax of the Java throw keyword is given below.

throw Instance i.e.,

1. throw new exception_class("error message");

Let's see the example of throw IOException.

1. throw new IOException("sorry device error");

Where the Instance must be of type Throwable or subclass of Throwable. For example, Exception is the sub class of Throwable and the user-defined exceptions usually extend the Exception class.

Java throw keyword Example

Example 1: Throwing Unchecked Exception

In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

TestThrow1.java

In this example, we have created the validate method that takes integer value as a

parameter. If the age is less than 18, we are throwing the ArithmeticException

otherwise print a message welcome to vote.

```
public class TestThrow1
{
//function to check if person is eligible to vote or not
```

```
public static void validate(int age)
{
 if(age<18)
 {
 //throw Arithmetic exception if not eligible to vote
 throw new ArithmeticException("Person is not eligible to vote");
 }
 else {
System.out.println("Person is eligible to vote!!");
 }
}
//main method
public static void main(String args[]){
//calling the function
validate(13);
System.out.println("rest of the code...");
}
}
```

Output:

The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

Example 2: Throwing Checked Exception

Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

TestThrow2.java

```
import java.io.*;

public class TestThrow2

{

//function to check if person is eligible to vote or not

public static void method() throws FileNotFoundException

{

FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
```

```
 BufferedReader fileInput = new BufferedReader(file);
12. throw new FileNotFoundException();
}
//main method
public static void main(String args[]){
try
{
method();
 }
 catch (FileNotFoundException e)
{
 e.printStackTrace();
}
 System.out.println("rest of the code...");
}
}
```

Output:

Example 3: Throwing User-defined Exception

exception is everything else under the Throwable class.

TestThrow3.java

```
// class represents user-defined exception
class UserDefinedException extends Exception
{
public UserDefinedException(String str)
{
// Calling constructor of parent Exception
super(str);
}
}
// Class that uses above MyException
public class TestThrow3
{
public static void main(String args[])
{
try
{
// throw an object of user defined exception
```

```
throw new UserDefinedException("This is user-defined exception");
}
catch (UserDefinedException ude)
{
System.out.println("Caught the exception");
// Print the message from MyException object
System.out.println(ude.getMessage());
}
}
}
```

Output:

File handling

File Operations in Java

In Java, a File is an abstract data type. A named location used to store related information is known as a File. There are several File Operations like creating a new File, getting information about File, writing into a File, reading from a File and deleting a File.

Before understanding the File operations, it is required that we should have knowledge of Stream and File methods. If you have knowledge about both of

them, you can skip it.

Stream

A series of data is referred to as a stream. In Java, Stream is classified into two types, i.e., Byte Stream and Character Stream.

Byte Stream Byte Stream is mainly involved with byte data. A file handling process with a byte stream is a process in which an input is provided and executed with the byte data.

Character Stream

Character Stream is mainly involved with character data. A file handling process with a character stream is a process in which an input is provided and executed with the character data.

To get more knowledge about the stream, click here.

Java File Class Methods

Type

Description

1. canRead() Boolean The canRead() method is used to check whether we can read the data of the file or not.

2. createNewFile() Boolean The createNewFile() method is used to create a new empty file.

3. canWrite() Boolean The canWrite() method is used to check whether we can write the data into the file or not.

4. exists() Boolean The exists() method is used to check whether the specified file is present or not.

5. delete() Boolean The delete() method is used to delete a file.

6. getName() String The getName() method is used to find the file name.

7. getAbsolutePath() String The getAbsolutePath() method is used to get the absolute pathname of the file.

8. length() Long The length() method is used to get the size of the file in bytes.

9. list() String[] The list() method is used to get an array of the files available in the directory.

10. mkdir() Boolean The mkdir() method is used for creating a new directory.

File Operations

We can perform the following operation on a file:

- Create a File
- Get File Information
- Write to a File
- Read from a File
- Delete a File

Create a File

Create a File operation is performed to create a new file. We use the createNewFile() method of file. The createNewFile() method returns

true when it successfully creates a new file and returns false when the file already exists.

Let's take an example of creating a file to understand how we can use

the createNewFile() method to perform this operation.

CreateFile.java

```
 // Importing File class

import java.io.File;

// Importing the IOException class for handling errors

 import java.io.IOException;

 class CreateFile

{

 public static void main(String args[])

{

 Try

 {

// Creating an object of a file

File f0 = new File("D:FileOperationExample.txt");

 if (f0.createNewFile())
```

```
{
 System.out.println("File " + f0.getName() + " is created successfully.");
 }
else {
 System.out.println("File is already exist in the directory.");
 }
 } catch (IOException exception) {
 System.out.println("An unexpected error is occurred.");
 exception.printStackTrace();
}
}
}
```

Explanation:

In the above code, we import the File and IOException class for performing file operation and handling errors, respectively. We create the f0 object of the File class and specify the location of the directory where we want to create a file. In the try block, we call the createNewFile() method through the f0 object to create a new file in the specified location. If the method returns false, it will jump to the else section.

If there is any error, it gets handled in the catch block.

Get File Information

The operation is performed to get the file information. We use several methods to get the information about the file like name, absolute path, is readable, is writable and length.

Let's take an example to understand how to use file methods to get the information

of the file.

FileInfo.java

```
// Import the File class
import java.io.File;
class FileInfo
{
public static void main(String[] args)
{
// Creating file object
File f0 = new File("D:FileOperationExample.txt");
if (f0.exists())
{
// Getting file name
```

```
System.out.println("The name of the file is: " + f0.getName());
// Getting path of the file
System.out.println("The absolute path of the file is: " +
f0.getAbsolutePath());
// Checking whether the file is writable or not
System.out.println("Is file writeable?: " + f0.canWrite());
// Checking whether the file is readable or not
System.out.println("Is file readable " + f0.canRead());
// Getting the length of the file in bytes
System.out.println("The size of the file in bytes is: " + f0.length());
}
else {
System.out.println("The file does not exist.");
}
}
}
```

Output:

Description:

In the above code, we import the java.io.File package and create a class FileInfo.

In the main method, we create an object of the text file which we have created in our

previous example. We check the existence of the file using a conditional statement, and if it is present, we get the following information about that file:

1. We get the name of the file using the getName()

2. We get the absolute path of the file using the getAbsolutePath() method of

the file.

3. We check whether we can write data into a file or not using the canWrite()

4. We check whether we can read the data of the file or not using the canRead()

5. We get the length of the file by using the length()

If the file doesn't exist, we show a custom message.

Write to a File

The next operation which we can perform on a file is "writing into a file". In order to write data into a file, we will use the FileWriter

class and its write() method together. We need to close the stream using the close() method to retrieve the allocated resources.

Let's take an example to understand how we can write data into a file.

WriteToFile.java

```
// Importing the FileWriter class
import java.io.FileWriter;
// Importing the IOException class for handling errors
import java.io.IOException;
class WriteToFile
{
public static void main(String[] args)
{
try
{
FileWriter fwrite = new FileWriter("D:FileOperationExample.txt");
// writing the content into the FileOperationExample.txt file
fwrite.write("A named location used to store related information is referred to as a File.");
```

```
// Closing the stream
fwrite.close();
System.out.println("Content is successfully wrote to the file.");
}
catch (IOException e)
{
System.out.println("Unexpected error occurred");
e.printStackTrace();
}
}
}
```

Output:

Explanation:

In the above code, we import

the java.io.FileWriter and java.io.IOException classes. We create a class WriteToFile, and in its main method, we use the try-catch block. In the try section, we create an instance of the FileWriter class, i.e., fwrite. We call the write method of the FileWriter class and pass the content to that function which we want to write.

After that, we call the close() method of the FileWriter class to close the file stream.

After writing the content and closing the stream, we print a custom message.

If we get any error in the try section, it jumps to the catch block. In the catch block, we handle the IOException and print a custom message.

Read from a File

The next operation which we can perform on a file is "read from a file". In order to write data into a file, we will use the Scanner class. Here, we need to close the stream using the close() method. We will create an instance of the Scanner class and use the hasNextLine() method nextLine() method to get data from the file.

Let's take an example to understand how we can read data from a file.

ReadFromFile.java

```
// Importing the File class

import java.io.File;

// Importing FileNotFoundException class for handling errors

import java.io.FileNotFoundException;

// Importing the Scanner class for reading text files

import java.util.Scanner;

class ReadFromFile

{
```

```
public static void main(String[] args)
{
try
{
// Create f1 object of the file to read data
File f1 = new File("D:FileOperationExample.txt");
Scanner dataReader = new Scanner(f1);
while (dataReader.hasNextLine())
{
String fileData = dataReader.nextLine();
System.out.println(fileData);
}
dataReader.close();
}
catch (FileNotFoundException exception)
{
System.out.println("Unexcpected error occurred!");
exception.printStackTrace();
}
```

```
 }

}
```

Output:

Expalnation:

In the above code, we import the "java.util.Scannner",

"java.io.File" and "java.io.IOException" classes. We create a class ReadFromFile, and in its main method, we use the try-catch block. In the try section, we create an instance of both the Scanner and the File classes. We pass the File class object to the Scanner class object and then iterate the scanner class object using the "While" loop and print each line of the file. We also need to close the scanner class object, so we use the close() function. If we get any error in the try section, it jumps to the catch block. In the catch block, we handle the IOException and print a custom message.

Delete a File

The next operation which we can perform on a file is "deleting a file". In order to delete a file, we will use the delete() method of the file. We don't need to close the stream using the close() method because for deleting a file, we neither use the FileWriter class nor the Scanner class.

Let's take an example to understand how we can write data into a file.

DeleteFile.java

```
// Importing the File class
```

```
import java.io.File;
class DeleteFile
{
public static void main(String[] args)
{
File f0 = new File("D:FileOperationExample.txt");
if (f0.delete())
{
System.out.println(f0.getName()+ " file is deleted successfully.");
}
else {
System.out.println("Unexpected error found in deletion of the file.");
}
}
}
```

Output:

Explanation:

In the above code, we import the File class and create a class DeleteFile. In the main() method of the class, we create f0 object of the file which we want to delete. In the if statement, we call the delete() method of the file using the f0 object. If the delete() method returns true, we print the success custom message. Otherwise, it jumps to the else section where we print the unsuccessful custom message. All the above-mentioned operations are used to read, write, delete, and create file programmatically.

UNIT 5

Applet, AWT and Swing Programming

Applet is a special type of program that is embedded in the webpage to generate the dynamic content. It runs inside the browser and works at client side.

Advantage of Applet

There are many advantages of applet. They are as follows:

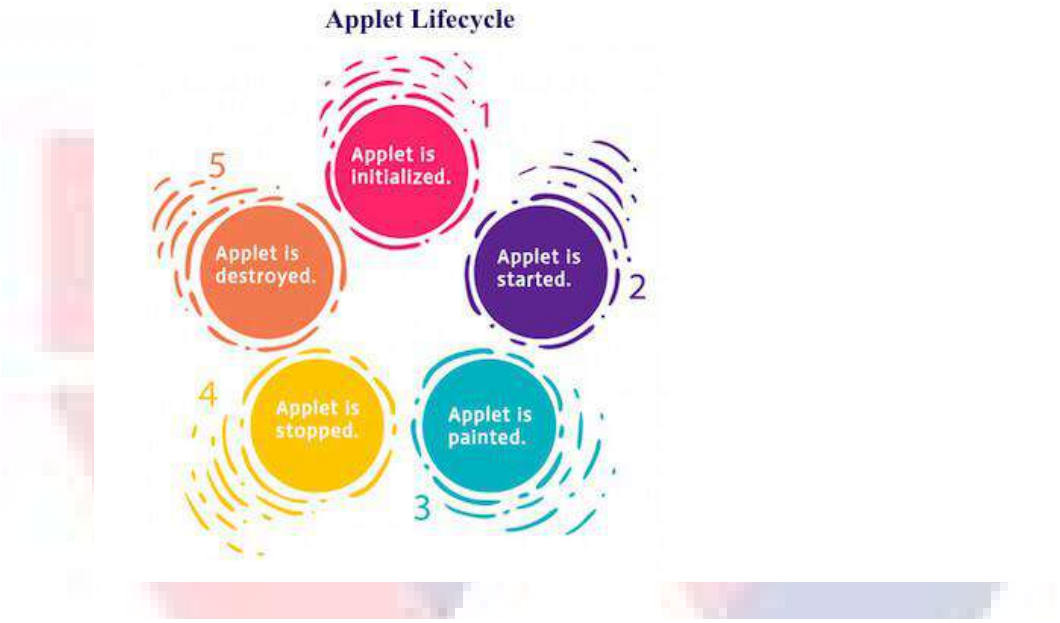
- It works at client side so less response time.
- Secured
- It can be executed by browsers running under many plateforms, including Linux, Windows, Mac Os etc.

Drawback of Applet

- Plugin is required at client browser to execute applet.

Lifecycle of Java Applet

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.



Lifecycle methods for Applet:

The java.applet.Applet class 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.

java.applet.Applet class

For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.

public void init(): is used to initialized the Applet. It is invoked only once.

public void start(): is invoked after the init() method or browser is maximized. It is used to start the Applet.

public void stop(): is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.

public void destroy(): is used to destroy the Applet. It is invoked only once.

java.awt.Component class

The Component class provides 1 life cycle method of applet.

public void paint(Graphics g): is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

How to run an Applet?

There are two ways to run an applet

By html file.

By appletViewer tool (for testing purpose).

Simple example of Applet by html file:

To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.

//First.java

```
import java.applet.Applet;

import java.awt.Graphics;

public class First extends Applet{

  public void paint(Graphics g){

g.drawString("welcome",150,150);

}

}
```

myapplet.html

```
<html>
<body>
<applet code="First.class" width="300" height="300">
</applet>
</body>
</html>
```

To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.

```
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{
  public void paint(Graphics g){
g.drawString("welcome to applet",150,150);
}
}
/*
<applet code="First.class" width="300" height="300">
```

</applet>

*/

To execute the applet by appletviewer tool, write in command prompt:

c:\>javac First.java

c:\>appletviewer First.java

Displaying Graphics in Applet

java.awt.Graphics class provides many methods for graphics programming.

Commonly used methods of Graphics class:

1. **public abstract void drawString(String str, int x, int y):** is used to draw the specified string.
2. **public void drawRect(int x, int y, int width, int height):** draws a rectangle with the specified width and height.
3. **public abstract void fillRect(int x, int y, int width, int height):** is used to fill rectangle with the default color and specified width and height.
4. **public abstract void drawOval(int x, int y, int width, int height):** is used to draw oval with the specified width and height.
5. **public abstract void fillOval(int x, int y, int width, int height):** is used to fill oval with the default color and specified width and height.
6. **public abstract void drawLine(int x1, int y1, int x2, int y2):** is used to draw line between the points(x1, y1) and (x2, y2).
7. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.

8. **public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used draw a circular or elliptical arc.
9. **public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used to fill a circular or elliptical arc.
10. **public abstract void setColor(Color c):** is used to set the graphics current color to the specified color.
11. **public abstract void setFont(Font font):** is used to set the graphics current font to the specified font.

```
import java.applet.Applet;
import java.awt.*;
  public class GraphicsDemo extends Applet{
  public void paint(Graphics g){
g.setColor(Color.red);
g.drawString("Welcome",50, 50);
g.drawLine(20,30,20,300);
g.drawRect(70,100,30,30);
g.fillRect(170,100,30,30);
g.drawOval(70,200,30,30);
  g.setColor(Color.pink);
g.fillOval(170,200,30,30);
```

```
g.drawArc(90,150,30,30,30,270);

g.fillArc(270,150,30,30,0,180);

 }

}
```

myapplet.html

```
<html>
<body>
<applet code="GraphicsDemo.class" width="300" height="300">
</applet>
</body>
</html>
```

AWT

Java AWT (Abstract Window Toolkit) is an API to develop Graphical User Interface (GUI) or windows-based applications in Java.Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavy weight i.e. its components are using the resources of underlying operating system (OS).The java.awt package provides classes for AWT API such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.

The AWT tutorial will help the user to understand Java GUI programming in simple and easy steps.

Java AWT calls the native platform calls the native platform (operating systems) subroutine for creating API components like TextField, ChechBox, button, etc.For example, an AWT GUI with components like TextField, label and button will have different look and feel for the

different platforms like Windows, MAC OS, and Unix. The reason for this is the platforms have different view for their native components and AWT directly calls the native subroutine that creates those components.

There are four types of containers in Java AWT:

1. Window
2. Panel
3. Frame
4. Dialog

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. We need to create an instance of Window class to create this container.

Panel

The Panel is the container that doesn't contain title bar, border or menu bar. It is generic container for holding the components. It can have other components like button, text field etc. An instance of Panel class creates a container, in which we can add components.

Frame

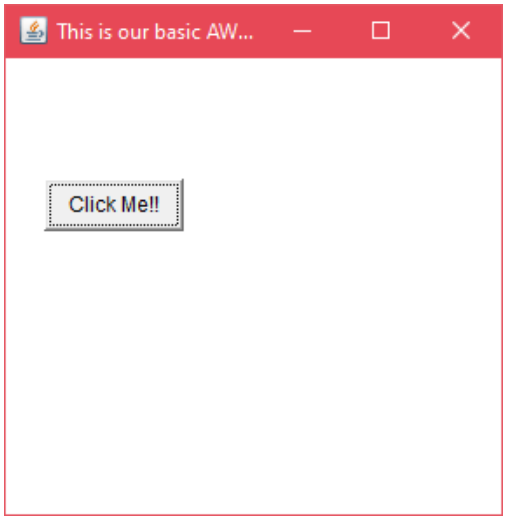
The Frame is the container that contain title bar and border and can have menu bars. It can have other components like button, text field, scrollbar etc. Frame is most widely used container while developing an AWT application.

AWTExample1.java

```
// importing Java AWT class
import java.awt.*;
  // extending Frame class to our class AWTExample1
public class AWTExample1 extends Frame {
     // initializing using constructor
   AWTExample1()
 {
        // creating a button
      Button b = new Button("Click Me!!");
        // setting button position on screen
      b.setBounds(30,100,80,30);
        // adding button into frame
      add(b);
        // frame size 300 width and 300 height
      setSize(300,300);
        // setting the title of Frame
      setTitle("This is our basic AWT example");
```

```
    // no layout manager
    setLayout(null);
     // now frame will be visible, by default it is not visible
    setVisible(true);
}

// main method
public static void main(String args[]) {
  // creating instance of Frame class
AWTExample1 f = new AWTExample1();
  }
}
```

Output:



Event Handling

Registration Methods

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){}

```
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();
}
}
```

1 LayoutManagers

03 The LayoutManagers are used to arrange components in a particular manner. The **Java LayoutManagers** facilitates us to control the positioning and size of the components in GUI forms. LayoutManager is an interface that is implemented by all the classes of layout managers. There are the following classes that represent the layout managers:

java.awt.BorderLayout

java.awt.FlowLayout

java.awt.GridLayout

java.awt.CardLayout

java.awt.GridBagLayout

javax.swing.BoxLayout

javax.swing.GroupLayout

javax.swing.ScrollPaneLayout

javax.swing.SpringLayout etc.

S Java BorderLayout

The BorderLayout is used to arrange the components in five regions: north, south, east, west, and center. Each region (area) may contain one component only. It is the default layout of a frame or window. The BorderLayout provides five constants for each region:

public static final int NORTH

public static final int SOUTH

public static final int EAST

public static final int WEST

public static final int CENTER

Constructors of BorderLayout class:

- **BorderLayout():** creates a border layout but with no gaps between the components.
- **BorderLayout(int hgap, int vgap):** creates a border layout with the given horizontal and vertical gaps between the components.

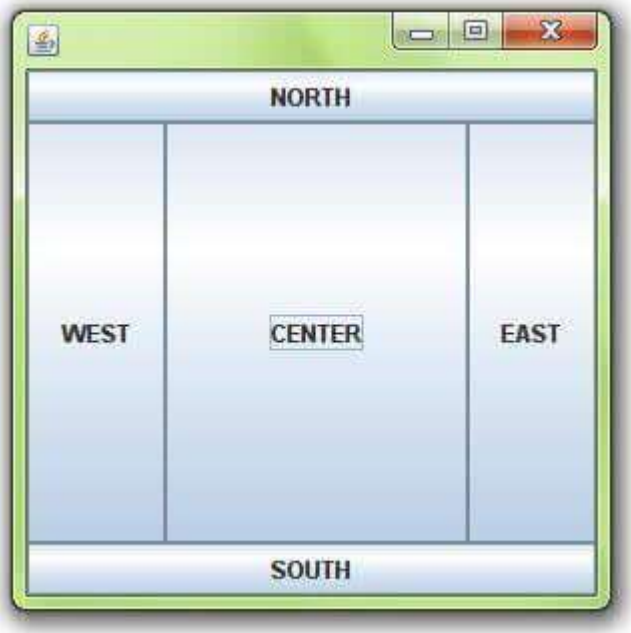
FileName: Border.java

```
import java.awt.*;
import javax.swing.*;
```

```
public class Border
{
JFrame f;
Border()
{
 f = new JFrame();

 // creating buttons
JButton b1 = new JButton("NORTH");; //the button will be labeled as NORTH

JButton b2 = new JButton("SOUTH");; // the button will be labeled as SOUTH

JButton b3 = new JButton("EAST");; // the button will be labeled as EAST

 JButton b4 = new JButton("WEST");; // the button will be labeled as WEST

JButton b5 = new JButton("CENTER");; // the button will be labeled as CENTER

f.add(b1, BorderLayout.NORTH); // b1 will be placed in the North Direction

f.add(b2, BorderLayout.SOUTH);  // b2 will be placed in the South Direction

 f.add(b3, BorderLayout.EAST);  // b2 will be placed in the East Direction
f.add(b4, BorderLayout.WEST);  // b2 will be placed in the West Direction

  f.add(b5, BorderLayout.CENTER);  // b2 will be placed in the Center

  f.setSize(300, 300);
    f.setVisible(true);
}
public static void main(String[] args) {
```

```
    new Border();
}
}
```

Output:



wings Rahul
102 SS

101 **Swing**

It 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.

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

}
```

JComponent

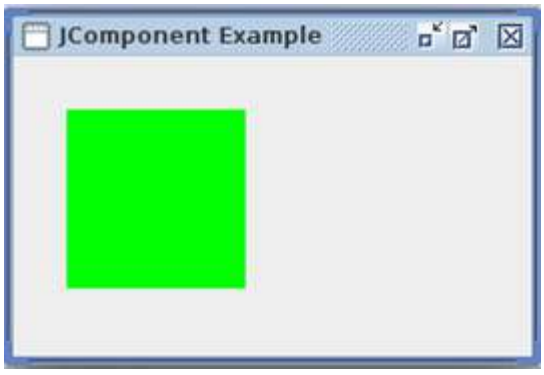
The JComponent class is the base class of all Swing components except top-level containers. Swing components whose names begin with "J" are descendants of the JComponent class. For example, JButton, JScrollPane, JPanel, JTable etc. But, JFrame and JDialog don't inherit JComponent class because they are the child of top-level containers.

The JComponent class extends the Container class which itself extends Component. The Container class has support for adding components to the container.

```
import java.awt.Color;
import java.awt.Graphics;
import javax.swing.JComponent;
import javax.swing.JFrame;
class MyJComponent extends JComponent {
    public void paint(Graphics g) {
      g.setColor(Color.green);
      g.fillRect(30, 30, 100, 100);
    }
}
public class JComponentExample {
    public static void main(String[] arguments) {
      MyJComponent com = new MyJComponent();
      // create a basic JFrame
      JFrame.setDefaultLookAndFeelDecorated(true);
      JFrame frame = new JFrame("JComponent Example");
      frame.setSize(300,200);
      frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
// add the JComponent to main frame
        frame.add(com);
        frame.setVisible(true);
      }
}
```

Output:



Vijay
100 Amit