

JAVA PROGRAMMING NOTES(R19)

(A0507194)

**Rajeev Gandhi Memorial College of Engineering and
Technology
Department of Computer Science and Engineering**

II B.Tech –II SEM CSE (R19)

UNIT-1

Introduction To Java – Introduction to OOP, OOP Concepts, History of Java, Java buzzwords, How Java differs from C and C++, Structure of Java Program, data types, variables, constants, type conversion and casting, enumerated types, scope and life time of variables, operators, expressions , control statements, command line arguments , arrays.

Introduction to OOPS:

Acronym of OOP: Object Oriented Programming.

Languages like Pascal, C, FORTRAN, and COBOL are called procedure oriented programming languages. Since in these languages, a programmer uses procedures or functions to perform a task. When the programmer wants to write a program, he will first divide the task into separate sub tasks, each of which is expressed as functions/ procedures. This approach is called procedure oriented approach.

The languages like C++ and Java use classes and object in their programs and are called Object Oriented Programming languages. The main task is divided into several modules and these are represented as classes. Each class can perform some tasks for which several methods are written in a class. This approach is called Object Oriented approach.

Points to be noted:

- Focuses on objects and data
- It provides security.
- Bottom-Up approach.
- Implements real world entities like objects, classes etc.
- Fast and easier to execute.
- Code reusability

OOP Concepts:

There are 6 OOP Concepts:

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

Classes:

Classes and objects are the two main aspects of object-oriented programming.

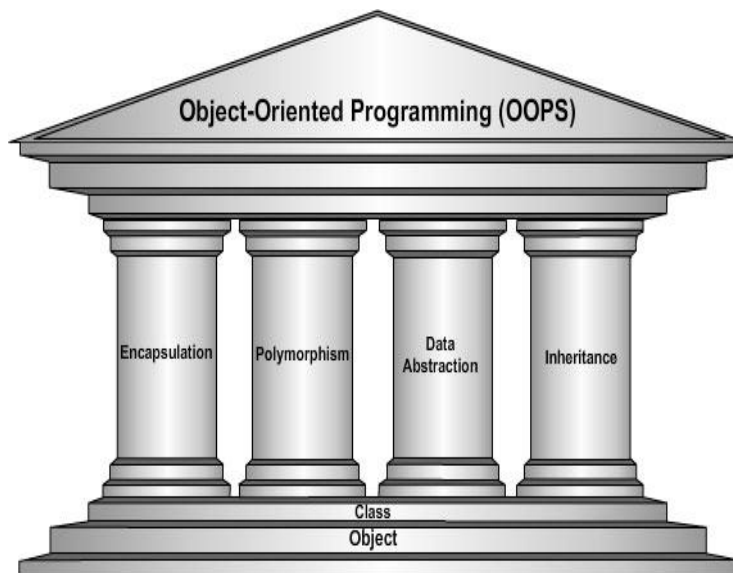
Class: It is a user-defined **blueprint** or prototype from which objects are created.

Or

A collection of objects is called Class.

- It represents the set of properties or methods that are common to all objects of one type.
- Class is a logical entity.
- It doesn't consume any space.

- Eg: Car, Fruit, Birds etc



In object-oriented programming, a class is a programming language construct that is used as a blueprint to create objects. This blueprint includes attributes and methods that the created objects all share. Usually, a class represents a person, place, or thing - it is an abstraction of a concept within a computer program. Fundamentally, it encapsulates the state and behavior of that which it conceptually represents. It encapsulates state through data placeholders called member variables; it encapsulates behavior through reusable code called methods.

Syntax:

class <class_name>

{

 Properties (variables);

 Actions (methods);

}

eg: **class** Student{

int rollNo; //properties -- variables

String name;

void display () { //methods –actions

 System.out.println ("Student Roll Number is: " + rollNo);

 System.out.println ("Student Name is: " + name);

 }

}

Note:

- Variables inside a class are called as instance variables.
- Variables inside a method are called as method variables.

Object:

- An instance of class is called object.

Or

- Any entity that has state and behavior is known as an object.
- An object contains an address and takes up some space in memory.

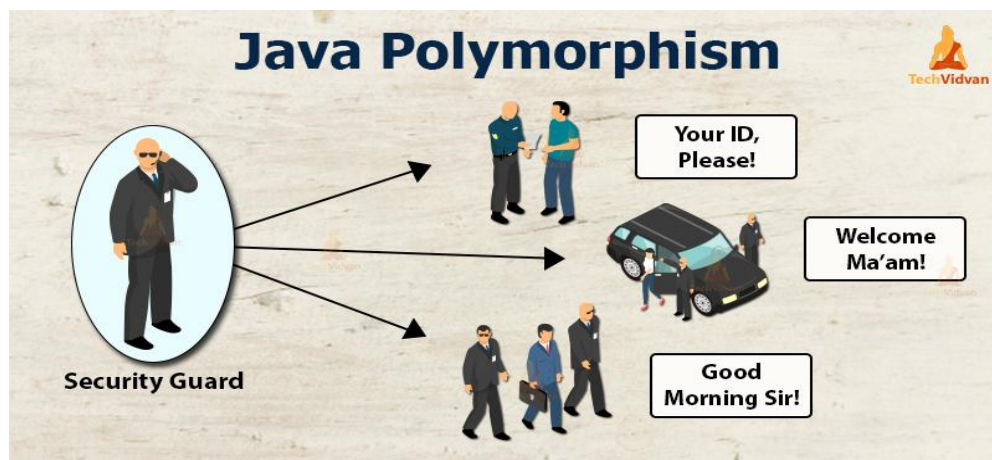
Ex:

Car
Rolls Royce
Audi
Ferrari

Polymorphism:

- If one task is performed in **different ways**, it is known as polymorphism.
- Polymorphism is a Greek Word i.e “Poly-**many** Morphism-**different forms**”.
- In Java, we use method overloading and method overriding to achieve polymorphism.

Ex:



Inheritance:

- When one object acquires all the properties and behaviors of a parent object, it is known as inheritance.
- Creates new class from existing class. Super class/Base class ---> Parent class, Sub class - --> Child class.

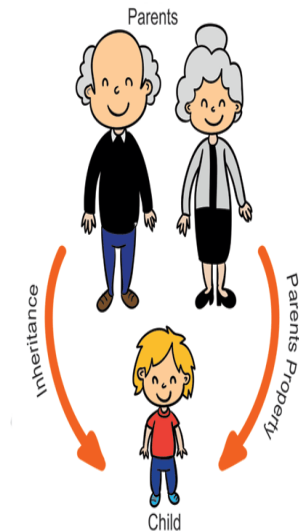
- “**extends**” keyword is used to inherit base class properties & methods to child class.
- It uses an IS-A relationship.
- **Types of inheritances:**
 1. Single
 2. Multilevel
 3. Hierarchical
 4. Multiple
 5. Hybrid

Note: Multiple inheritance in Java is not supported using classes.

Advantages:

1. Code reusability.
2. Runtime polymorphism(Method Overriding).

Ex:

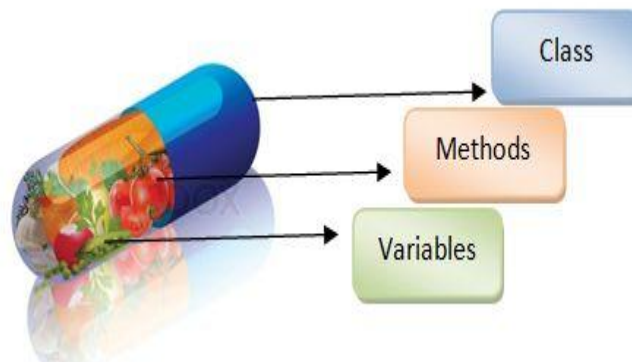


Encapsulation:

- Wrapping up of code and data in to sing unit is called encapsulation.

Advantages:

1. Data hiding.
 2. Easy to test.
- Ex: Java Bean.



Abstraction:

- Hiding the implementation details & showing only functionality.
- Can achieve by using abstract class & interface.

Advantages:

1. Reduces complexity.
 2. Increases security.
- Ex: ATM

History of Java:

Important points:

- **James Gosling, Chris Wart, Ed Frank, Mike Sheridan, and Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
- Bill Joy, Arthur Van Hoff, Jonathan Payne, Payne Yellin and Tim were key Contributors.
- From **C**, Java derives syntax & OOP features from **C++**.
- Firstly, it was named as “**Green Talk**” later as “**OAK**” by Green project.
- In 1995 renamed as “**JAVA**” because OAK is already a trademark of OAK tech.
- Initially developed by James Gosling at Sun Microsystems and released in 1991.
- Provides **WORA**(Write Once, Run Anywhere).
- **James Gosling** developed Suitable Language, **Patrick Naughton** developed Graphics, **Bill Joy** developed Web Browser.
- 1st version of Java 1.0 released on 1996.

Brief History:

In 1990, Sun Micro Systems Inc (US) was conceived a project to develop software for consumer electronic devices that could be controlled by a remote This project was called

Stealth Project but later its name was changed to Green Project. In January 1991, Project Manager James Gosling and his team members Patrick Naughton, Mike Sheridan, Chris Wrath, and Ed Frank met to discuss about this project. Gosling thought C and C++ would be used to develop the project. But the problem he faced with them is that they were system dependent languages. The trouble with C and C++ (and most other languages) is that they are designed to be compiled for a specific target and could not be used on various processors, which the electronic devices might use. James Gosling with his team started developing a new language, which was completely system independent. This language was initially called OAK. Since this name was registered by some other company, later it was changed to Java. James Gosling and his team members were consuming a lot of coffee while developing this language. Good quality of coffee was supplied from a place called "Java Island". Hence they fixed the name of the language as Java. The symbol for Java language is cup and saucer. Sun formally announced Java at Sun World conference in 1995. On January 23rd 1996, JDK10 version was released.

Java is a Platform **independent** language.

- Java is an Object Oriented Language, used to develop Internet applications.
- Provides security against eavesdropping, tampering, impersonation.

JVM(Java Virtual Machine): It is a specification that provides a runtime environment in which Java bytecode can be executed.

What is JVM?

1. It is a **A specification** where the working of Java Virtual Machine is specified. But the implementation provider is independent to choose the algorithm. Its implementation has been provided by Oracle and other companies.
2. It is **an implementation known** as JRE (Java Runtime Environment).
3. It is a **Runtime Instance**. Whenever you write java command on the command prompt to run the java class, an instance of JVM is created.

What it does

Main tasks of JVM are:

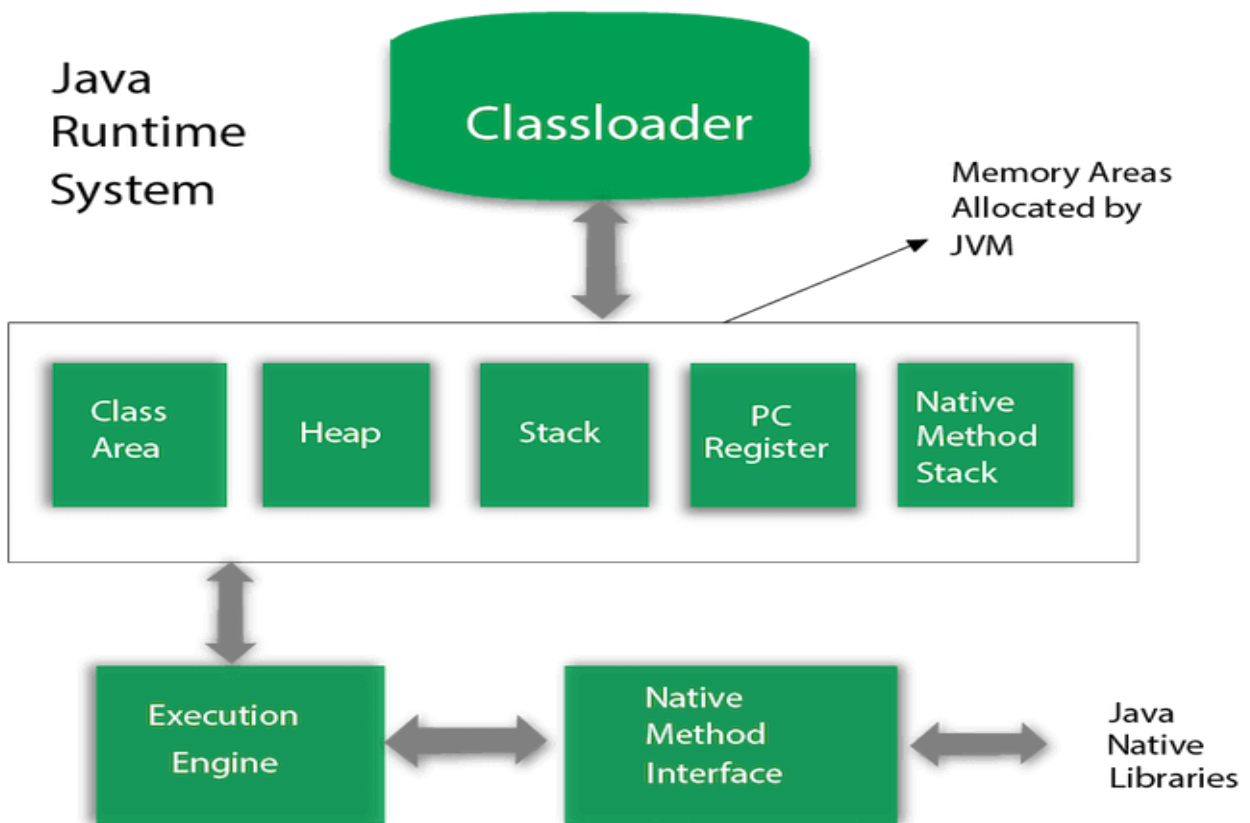
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

JVM provides definitions for the:

- Memory area
- Class file format
- Register set
- Garbage-collected heap

- Fatal error reporting etc.

JVM Architecture



Let's understand the internal architecture of JVM. It contains classloader, memory area, execution engine etc.

1. Classloader

First of all, the .java program is converted into a .class file consisting of byte code instructions by the Java compiler. Remember, this Java compiler is outside the JVM.

Now this class file is given to the JVM. In JVM, there is a module (or program) called **class loader subsystem**, which performs the following functions:

- First of all, it loads the .class file into memory.
- Then it verifies whether all byte code instructions are proper or not. If it finds any instruction suspicious, the execution is rejected immediately.
- Then it verifies whether all byte code instructions are proper or not. If it finds any instruction suspicious, the execution is rejected immediately.
- If the byte instructions are proper, then it allocates necessary memory to execute the program. This memory is divided into 5 parts, called run time data areas, which contain the data and results while running the program.

These areas are as follows:

Classloader is a subsystem of JVM which is used to load class files. Whenever we run the java program, it is loaded first by the classloader. There are three built-in classloaders in Java.

1. **Bootstrap ClassLoader:** This is the first classloader which is the super class of Extension classloader. It loads the *rt.jar* file which contains all class files of Java Standard Edition like java.lang package classes, java.net package classes, java.util package classes, java.io package classes, java.sql package classes etc.
2. **Extension ClassLoader:** This is the child classloader of Bootstrap and parent classloader of System classloader. It loads the jar files located inside *\$JAVA_HOME/jre/lib/ext* directory.
3. **System/Application ClassLoader:** This is the child classloader of Extension classloader. It loads the classfiles from classpath. By default, classpath is set to current directory. You can change the classpath using "-cp" or "-classpath" switch. It is also known as Application classloader.

These are the internal classloaders provided by Java. If you want to create your own classloader, you need to extend the ClassLoader class.

2. Class(Method) Area

- Method area is the memory block, which stores the class code, code of the
- variables, and code of the methods in the Java program. (Method means functions written in a class)
- Class(Method) Area stores per-class structures such as the runtime constant pool, field and method data, the code for methods.

3. Heap

- This is the area where objects are created. Whenever JVM loads a class, a method and a heap area are immediately created in it.
- It is the runtime data area in which objects are allocated.

4. Stack

- Method code is stored on Method area. But while running a method, it needs some more memory to store the data and results. This memory is allotted on Java stacks.
- So, Java stacks are memory areas where Java methods are executed. While executing methods, a separate frame will be created in the Java stack, where the method is executed. JVM uses a separate thread (or process) to execute each method.

5. Program Counter Register

These are the registers (memory areas), which contain memory address of the instructions of the methods. If there are 3 methods, 3 PC registers will be used to track the instructions of the methods.

6. Native Method Stack

Java methods are executed on Java stacks. Similarly, native methods (for example C/C++ functions) To execute the native are executed on Native method stacks. methods, generally native method libraries (for example C/C++ header files) are required. These header files are located and connected to JVM by a program, called Native method interface.

7. Execution Engine

- Execution engine contains an interpreter and **JIT** (Just In Time) compiler, which are responsible for converting the byte code instructions into machine code so that the processor will execute them.
- Most of the JVM implementations use both the interpreter and JIT compiler simultaneously to convert the byte code. This technique is also called **adaptive optimizer**.

It contains:

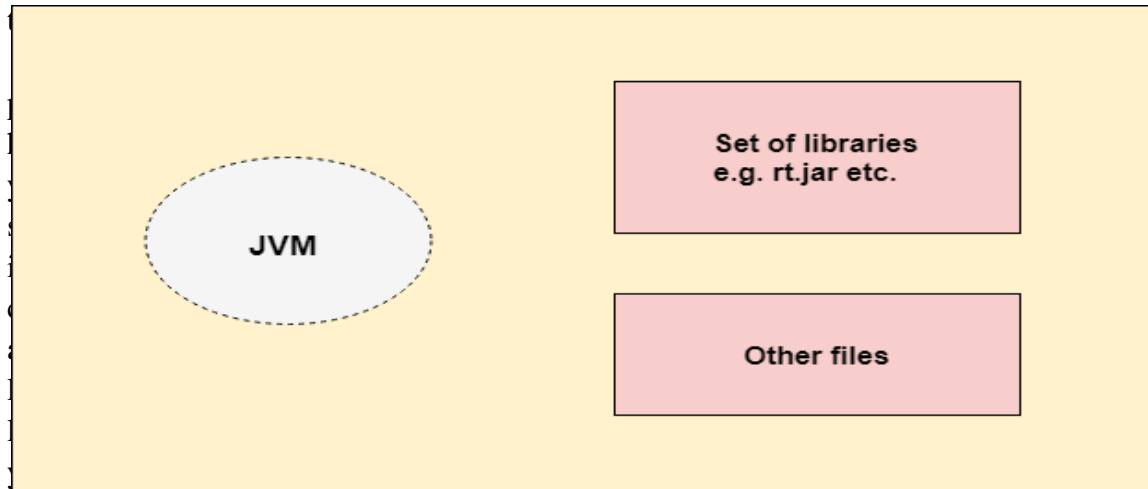
1. **A virtual processor**
2. **Interpreter:** Read bytecode stream then execute the instructions.
3. **Just-In-Time(JIT) compiler:** It is used to improve the performance. JIT compiles parts of the byte code that have similar functionality at the same time, and hence reduces the amount of time needed for compilation. Here, the term "compiler" refers to a translator from the instruction set of a Java virtual machine (JVM) to the instruction set of a specific CPU.

8. Java Native Interface

Java Native Interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++, Assembly etc. Java uses JNI framework to send output to the Console or interact with OS libraries.

JRE(Java Runtime Environment):

- 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.
- I



JRE

e
xists.

It also includes:

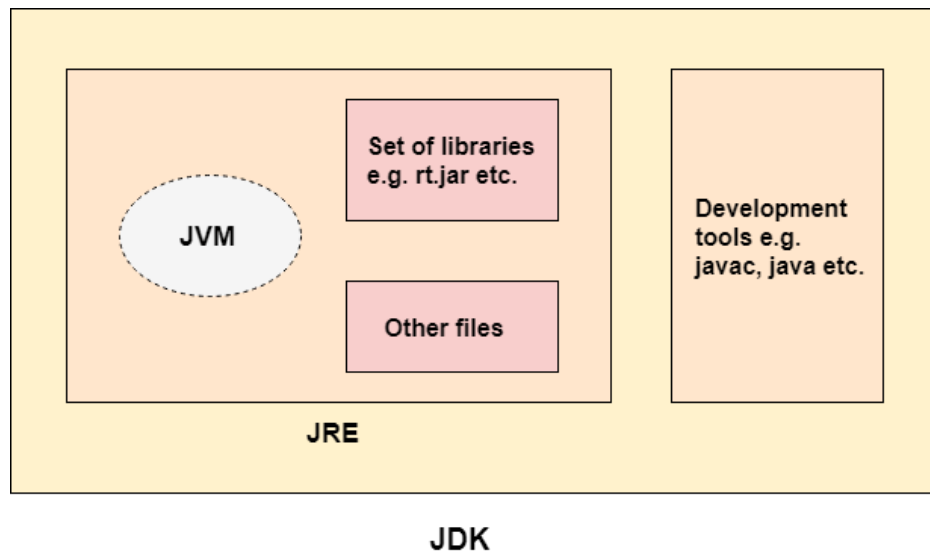
- Technologies which get used for deployment such as Java Web Start.
- Toolkits for user interface like Java 2D.
- Integration libraries like Java Database Connectivity (JDBC) and Java Naming and Directory Interface (JNDI).
- Libraries such as Lang and util.
- Other base libraries like Java Management Extensions (JMX), Java Native Interface (JNI)

and Java for XML Processing (JAX-WS).

JDK(Java Development Kit)

- Used to develop Java Applications & Applets.
- It physically exists.
- JDK is an implementation of any one of the below given Java Platforms.
 - Standard Edition Java Platform
 - Enterprise Edition Java Platform

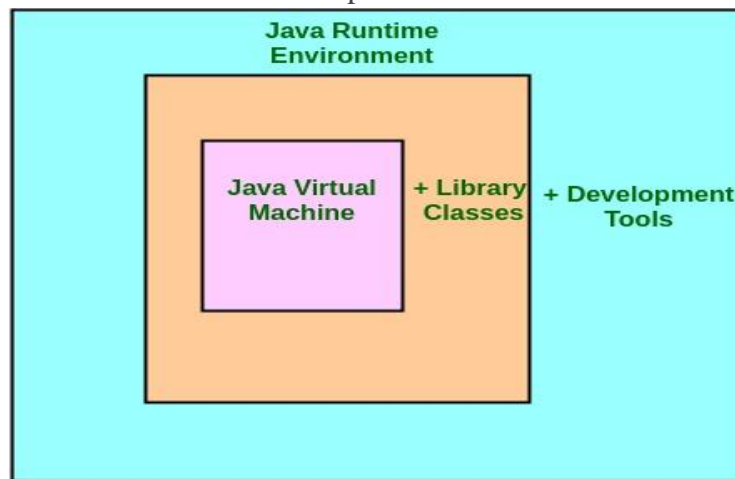
○ Micro
Edition Java Platform



Differences between JDK, JRE, JVM

- **JDK** – Java Development Kit (in short JDK) is Kit which provides the environment to develop and execute(run) the Java program. JDK is a kit(or package) which includes two things
 1. Development Tools(to provide an environment to develop your java programs)
 2. JRE (to execute your java program).
- **Note** : JDK is only used by Java Developers.
- **JRE** – Java Runtime Environment (to say JRE) is an installation package which provides environment to only run(not develop) the java program(or application) onto your machine. JRE is only used by them who only wants to run the Java Programs i.e. end users of your system.
- **JVM** – Java Virtual machine(JVM) is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by

line hence it is also known as interpreter



JDK = JRE + Development Tool
JRE = JVM + Library Classes

Java Installation:

Step 1: Go to [link](#). Click on JDK Download for Java JDK 8 download.

Java SE 8

Java SE 8u271 is the latest release for the Java SE 8 Platform.






- Documentation
- Installation Instructions
- Release Notes
- Oracle License
 - Binary License
 - Documentation License
 - BSD License
- Java SE Licensing Information User Manual
 - Includes Third Party Licenses
- Certified System Configurations
- Readme Files
 - JDK ReadMe
 - JRE ReadMe

Oracle JDK

- ↓ [JDK Download](#)
- ↓ [Server JRE Download](#)
- ↓ [JRE Download](#)
- ↓ [Documentation Download](#)
- ↓ [Demos and Samples Download](#)

Step 2: Next,

- Accept License Agreement
- Download Java 8 JDK for your version 32 bit or JDK 8 download for windows 10 64 bit.

Solaris SPARC 64-bit	88.75 MB	 jdk-8u271-solaris-sparcv9.tar.gz
Solaris x64 (SVR4 package)	134.42 MB	 jdk-8u271-solaris-x64.tar.Z
Solaris x64	92.52 MB	 jdk-8u271-solaris-x64.tar.gz
Windows x86	154.48 MB	 jdk-8u271-windows-i586.exe
Windows x64	166.79 MB	 jdk-8u271-windows-x64.exe

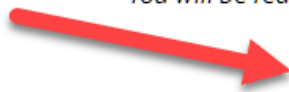
Step 3:

When you click on the Installation link the popup will be open. Click on I reviewed and accept the Oracle Technology Network License Agreement.

You must accept the [Oracle Technology Network License Agreement for Oracle Java SE](#) to download this software. ✕

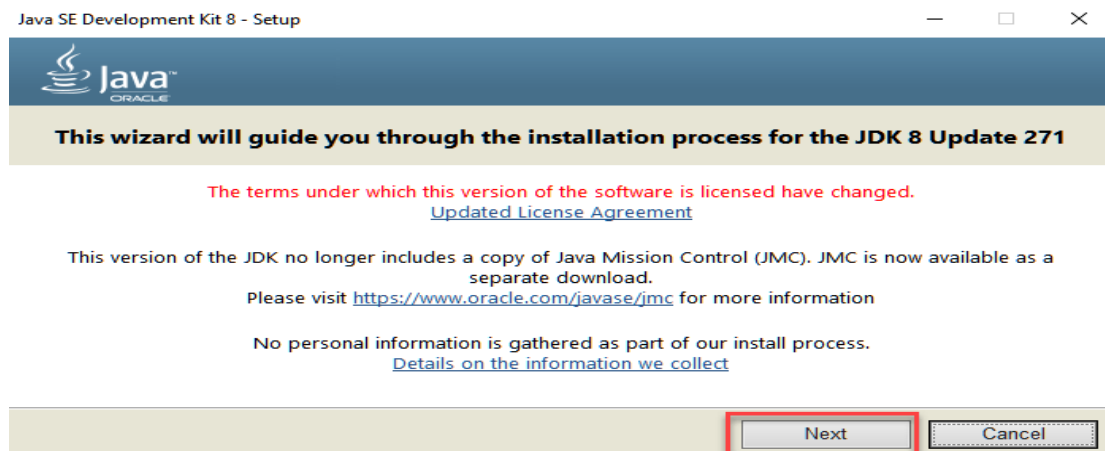
☒ I reviewed and accept the Oracle Technology Network License Agreement for Oracle Java SE

You will be redirected to the login screen in order to download the file.

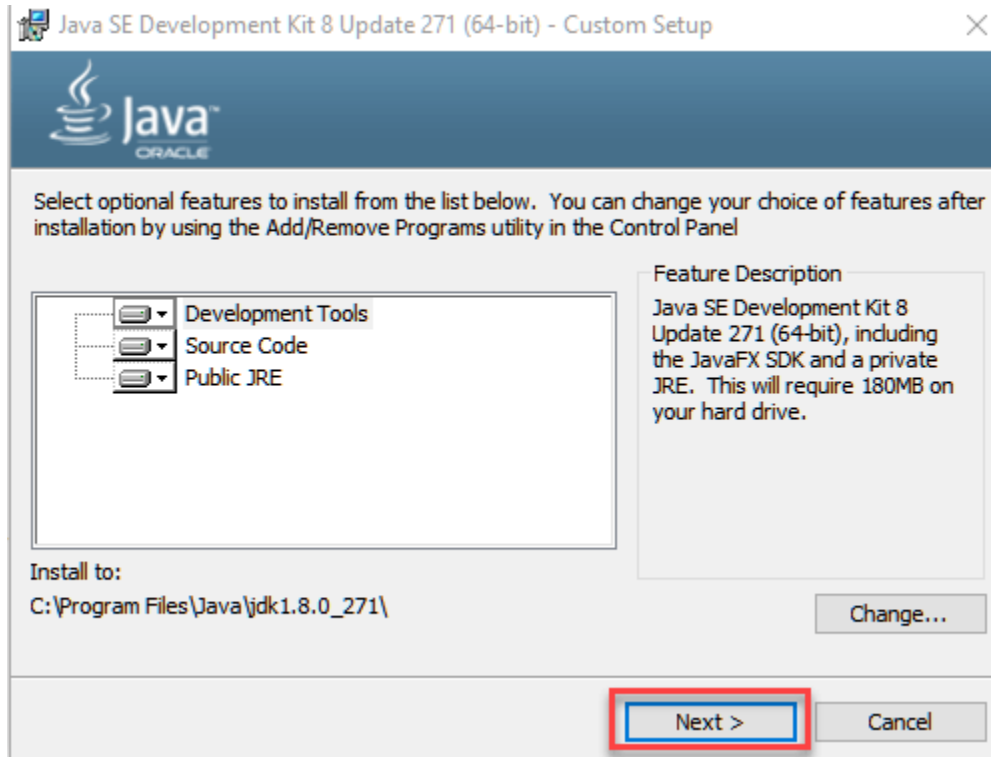


Download [jdk-8u271-windows-x64.exe](#) 

Step 4: Once the Java JDK 8 download is complete, run the exe for install JDK. Click Next



Step 5: Select the PATH to install Java in Windows... You can leave it Default. Click next.



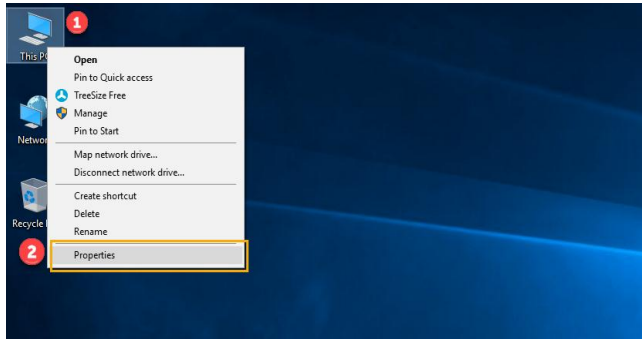
Step 6: Once you install Java in windows, click Close



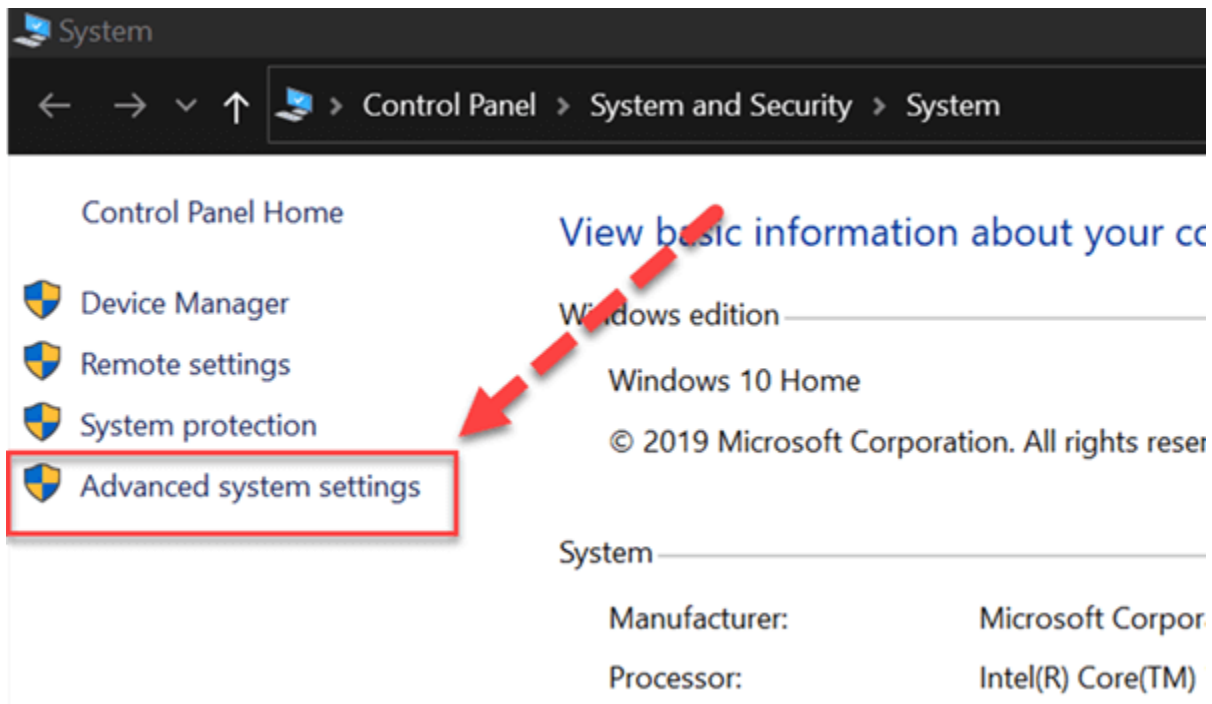
How to set Environment Variables in Java: Path and Classpath

The CLASSPATH variable gives location of the Library Files.

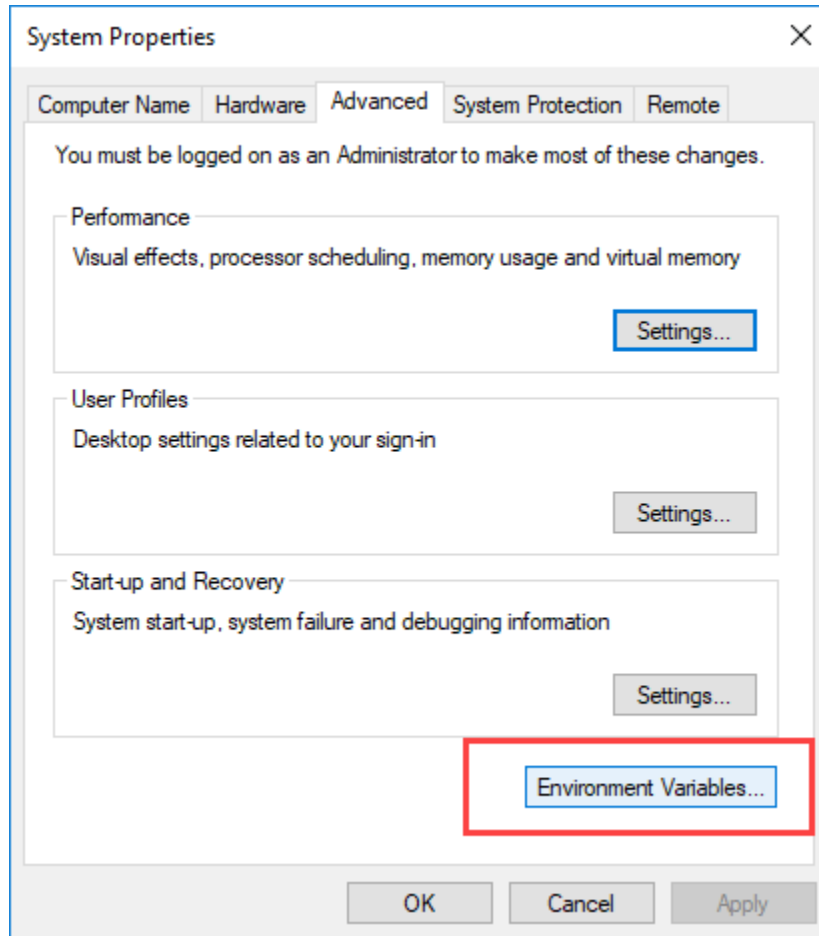
Step 1: Right Click on the My Computer and Select the properties



Step 2: Click on advanced system settings

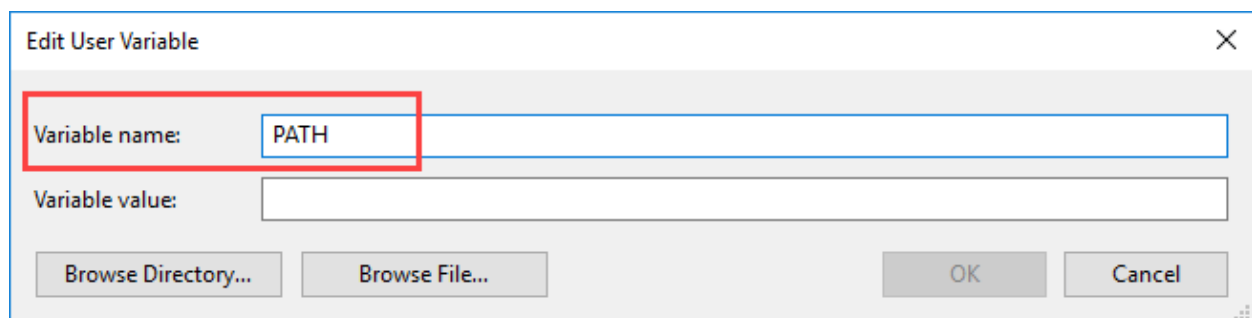


Step 3: Click on Environment Variables

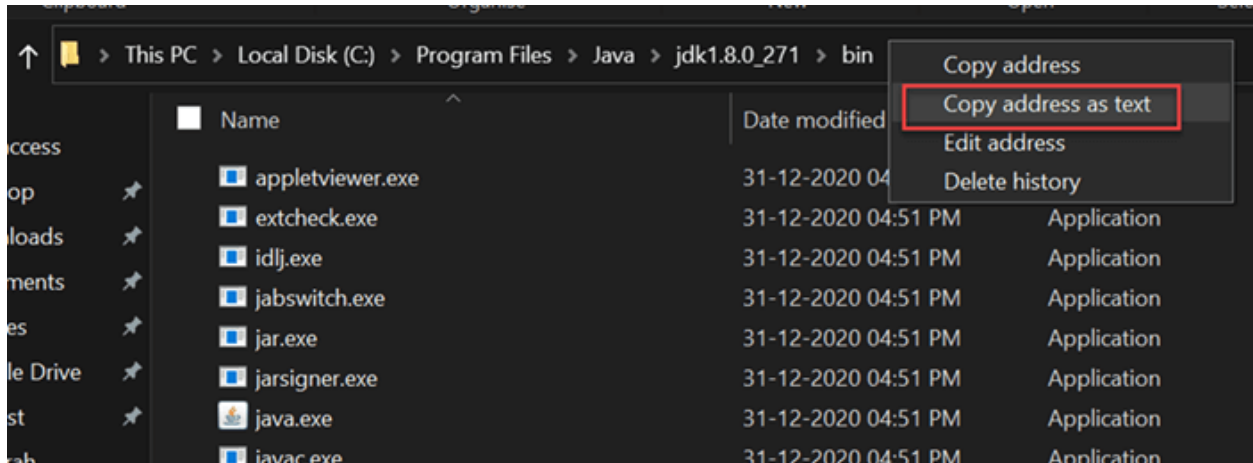


Step 4 Click on new Button of User variables

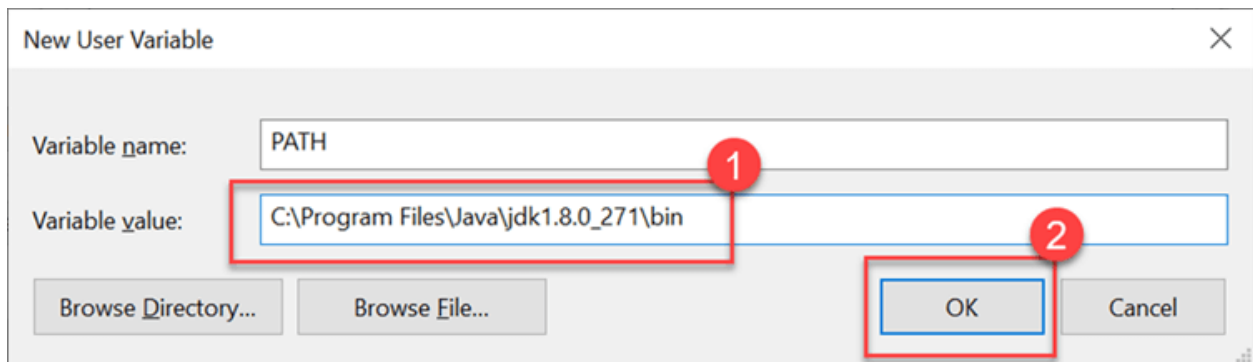
Step 5: Type PATH in the Variable name.



Step 6: Copy the path of the bin folder which is installed in JDK folder.

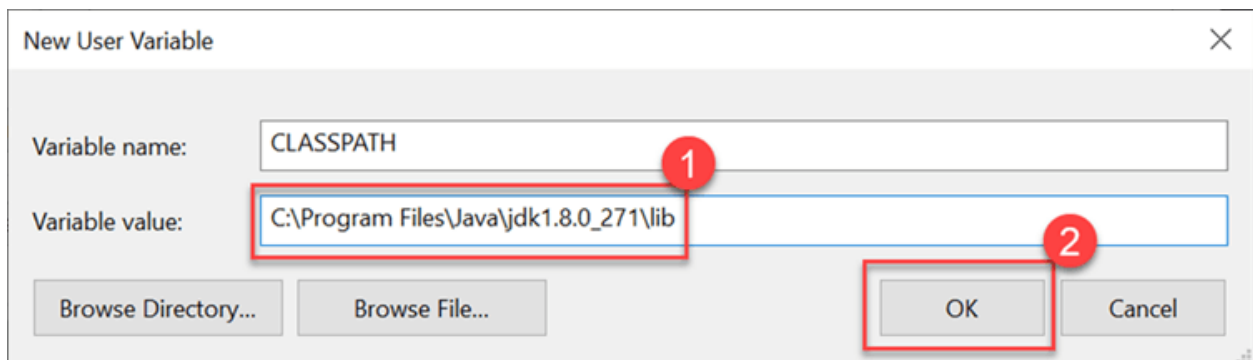


Step 7: Paste Path of bin folder in Variable value. Click on OK Button.

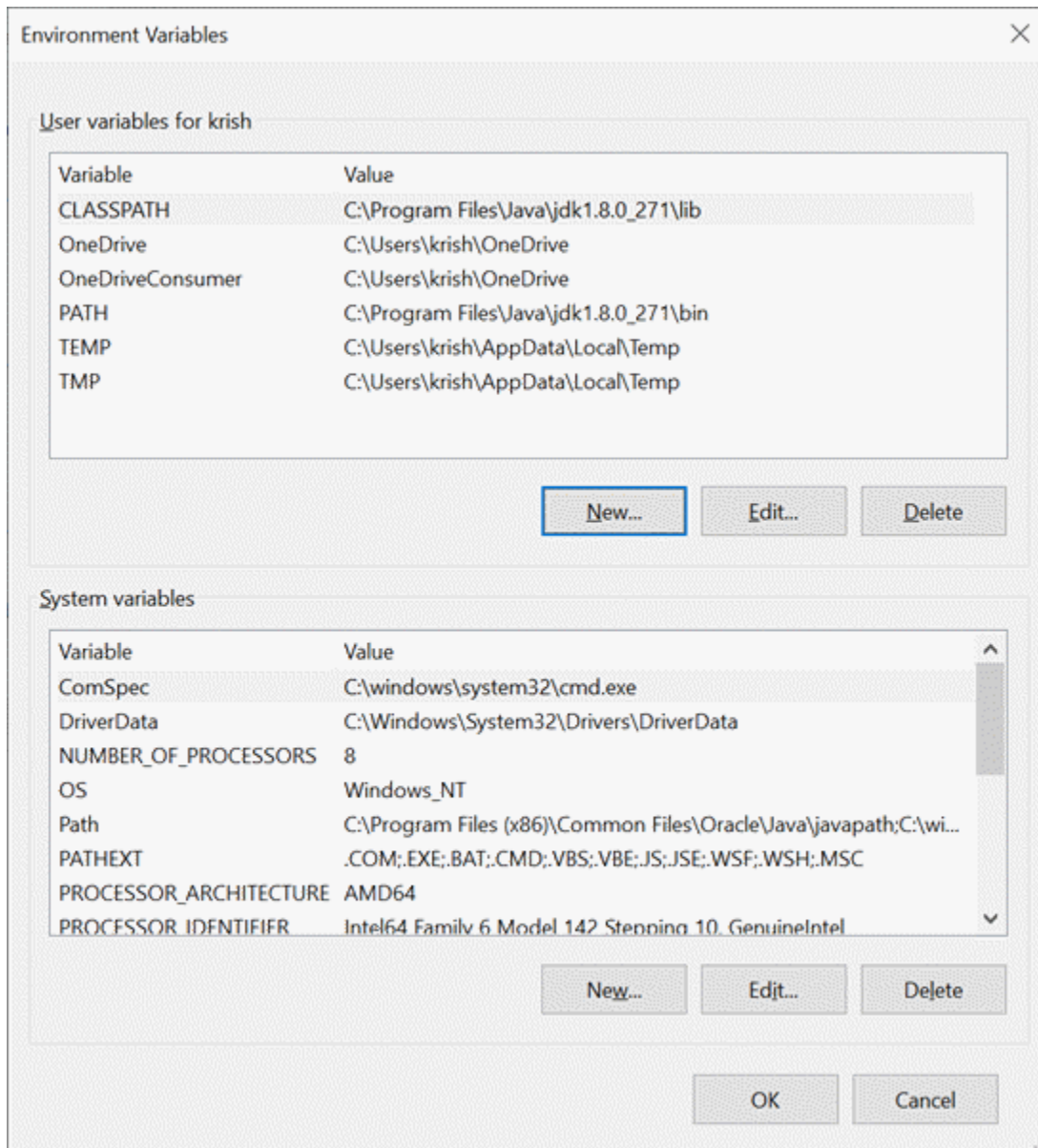


Note: In case you already have a PATH variable created in your PC, edit the PATH variable to **PATH = <JDK installation directory>\bin;%PATH%;**

Step 8: You can follow a similar process to set CLASSPATH.



Step 9: Click on OK button



Step 10: Go to command prompt and type **javac** command. List of options are displayed on successful installation.

How Java Differ from C & C++.

Comparison Index	C++	Java
Platform-independent	C++ is platform-dependent.	Java is platform-independent.
Goto	C++ supports the goto statement.	Java doesn't support the goto statement.
Multiple inheritance	C++ supports multiple inheritance.	Java doesn't support multiple inheritance through class. It can be achieved by interfaces in java.
Operator Overloading	C++ supports operator overloading.	Java doesn't support operator overloading.
Pointers	C++ supports pointers.	Java supports pointer internally.

Comparison Index	C++	Java
Compiler and Interpreter	C++ uses compiler only.	Java uses compiler and interpreter both.
Call by Value and Call by reference	C++ supports both call by value and call by reference.	Java supports only call by value only.
Structure and Union	C++ supports structures and unions.	Java doesn't support structures and unions.
Thread Support	C++ doesn't have built-in support for threads.	Java has built-in thread support.
Virtual Keyword	C++ supports virtual keyword .	Java has no virtual keyword.
Object-oriented	C++ is an object-oriented language.	Java is also an object oriented language.

C	Java
C is a Procedural Programming Language.	Java is an Object-Oriented language.
C was developed by Dennis M. Ritchie in 1972.	Java language was developed by James Gosling in 1995.
It is a middle-level language as it is binding the gaps between machine level and high-level languages.	It is a high-level language because the translation of code takes place into machine language, which uses a compiler or interpreter.
In the C declaration variables are declared at the beginning of the block.	In Java, you can declare a variable anywhere.
Free is a variable used for freeing the memory in C.	A compiler will free up the memory by calling the garbage collector.
C does not support threading.	Java has a feature of threading.
C support pointers.	Java does not support pointers.
Memory allocation can be done by malloc.	Memory allocation can be done by a new keyword.
Garbage collectors need to manage manually.	In Java, it is automatically managed by a garbage collector
C does not have a feature of overloading functionality.	Java supports method overloading.

C offers support for call by value and call by reference. Java only supports a call by value reference.

Structure of Java Program:

- Comment level section.
- Package section.
- Import section.
- Class or interface section.
- Class with main method.

Comment level section:

Java has 3 types of comments:

- Single line comments. (`// Single line comment`)
- Multiline comments. (`/* Hello multiline`

-----*/)

- Document level comments. (`/** javadoc provides documentation level comments*/`)

Package level section:

- Collection of similar types packages, interfaces and subpackages.
- Two types of packages:
 1. User-defined packages.
 2. Predefined packages/Built-in package.
- Built-in packages are java, lang, awt, javax, io, util, etc.
- **package** is a keyword in java

Syntax: package package-name;

Ex: package mypack;

Import section:

- **import** is java keyword.
- The **import statement** can be **used** to **import** an entire package or sometimes **import** certain classes and interfaces inside the package.

Syntax: import package-name;

Ex: import java.util.*;

Class or interface section:

- We can create an **interface** or **class** in this section if required. We use the **interface** or **class** keyword to create an interface or class respectively.

Syntax: interface interface_name{---}

Ex: **interface** abc{
 void start();
}

Syntax: class class_name{---}

Ex: **class** Student {
 int a;
}

Class with main method:

- we define the **main() method**. It is essential for all Java programs. Because the execution of all Java programs starts from the main() method.
- This method must be inside the class. Here we create objects for class and call methods.

Syntax: **public static void** main(String args[])

Ex: **class** Student {
 public static void main(String args[]){
 //stmts
 }
}

Simple Java Program:

- **Hello.java**

Ex: **public class** Hello {
 public static void main(String args[]){
 System.out.println("Hello World");
 }
}

Compilation: javac Hello.java

Execution: java Hello

Different Notations of main():

- **public static void** main(String args[]) //default prototype
- **static public void** main(String[] args) //swap positions

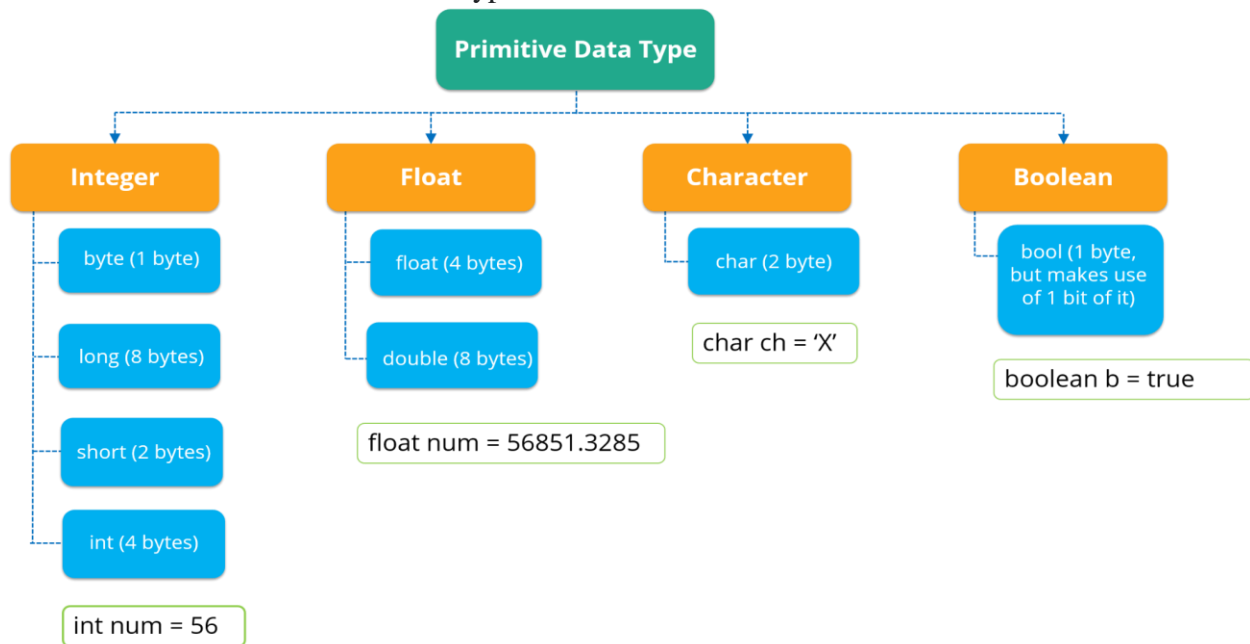
- **variant of string array argument:**
 - **public static void** main(String[] args)
 - **public static void** main(String []args)
- **public static void** main(String[] hello) //Instead of args we can write anything which is a valid java identifier.
- **final public static void** main(String... args) //var args
- **public static void** main(**final** String[] args) //final modifier
- **final public static void** main(String args[]) //final method
- **public synchronized static void** main(String[] args) //synchronized keyword
- **public strictfp static void** main(String[] args) //to restrict floating-point calculations
- **final synchronized strictfp public static void** main(String[] hello) //combination of all
- main() can be overloaded.

Naming Conventions in Java:

- The class/variable/method names must not contain any white space.
- No special characters (!@#%^&*) are allowed to use in class/variable/method names except \$ and _.
- Class/interface name should start with uppercase letters.
 - if the name contains multiple words, the 1st letter of the word must start with an uppercase letter such as **HelloWorld**.
- Method names should start with lowercase letters. **Ex. start(), stop(), run(), sleep(), etc.**
 - If the name contains multiple words, start it with a lowercase letter followed by an uppercase letter such as **actionPerformed(), nextLine(), nextInt(), parseInt(), etc.**
- Variable names must start with lowercase, special characters are not allowed.
 - If the name contains multiple words, start it with the lowercase letter followed by an uppercase letter such as **firstName, lastName**.
- Package name should be a lowercase letter such as java, lang.
 - If the name contains multiple words, it should be separated by dots (.) such as **java.util, java.lang**.
- Constant should be in uppercase letters such as **RED, YELLOW**.
 - If the name contains multiple words, it should be separated by an underscore(_) such as **MAX_PRIORITY**.
 - It may contain digits but not as the first letter.

Data Types:

- Data types specify the different sizes and values that can be stored in the variable.
- There are 2 types of data type:
 1. Primitive Data Types. (boolean, char, byte, short, long, int, float, double)
 2. Non-Primitive Data Types. (Classes, Interfaces, Arrays)
- **Why is Java a strongly typed language?**
 - **Java** is a **strongly typed** programming **language** because every variable must be declared with a data type.



Boolean Data Type:

- The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.
- The Boolean data type specifies one bit of information, but its "size" can't be defined precisely.

Example: Boolean one = false

Byte Data Type:

- The byte data type is an example of primitive data type. It is an 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.
- The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of "int" data type.

Example: byte a = 10, byte b = -20

Short Data Type:

- The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.
- The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.

Example: short s = 10000, short r = -5000

Int Data Type:

- The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^{31}) to 2,147,483,647 ($2^{31} - 1$) (inclusive). Its minimum value is - 2,147,483,648 and maximum value is 2,147,483,647. Its default value is 0.
- The int data type is generally used as a default data type for integral values unless there is no problem about memory.

Example: int a = 100000, int b = -200000

Long Data Type:

The long data type is a 64-bit two's complement integer. Its value-range lies between (-2^{63}) to $(2^{63} - 1)$ (inclusive). Its minimum value is - 9,223,372,036,854,775,808 and maximum value is 9,223,372,036,854,775,807. **Its default value is 0.** The long data type is used when you need a range of values more than those provided by int.

Example: long a = 100000L, long b = -200000L

Float Data Type:

The float data type is a single-precision 32-bit IEEE 754 floating point. Its value range is unlimited. It is recommended to use a float (instead of double) if you need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

Example: float f1 = 234.5f

Double Data Type:

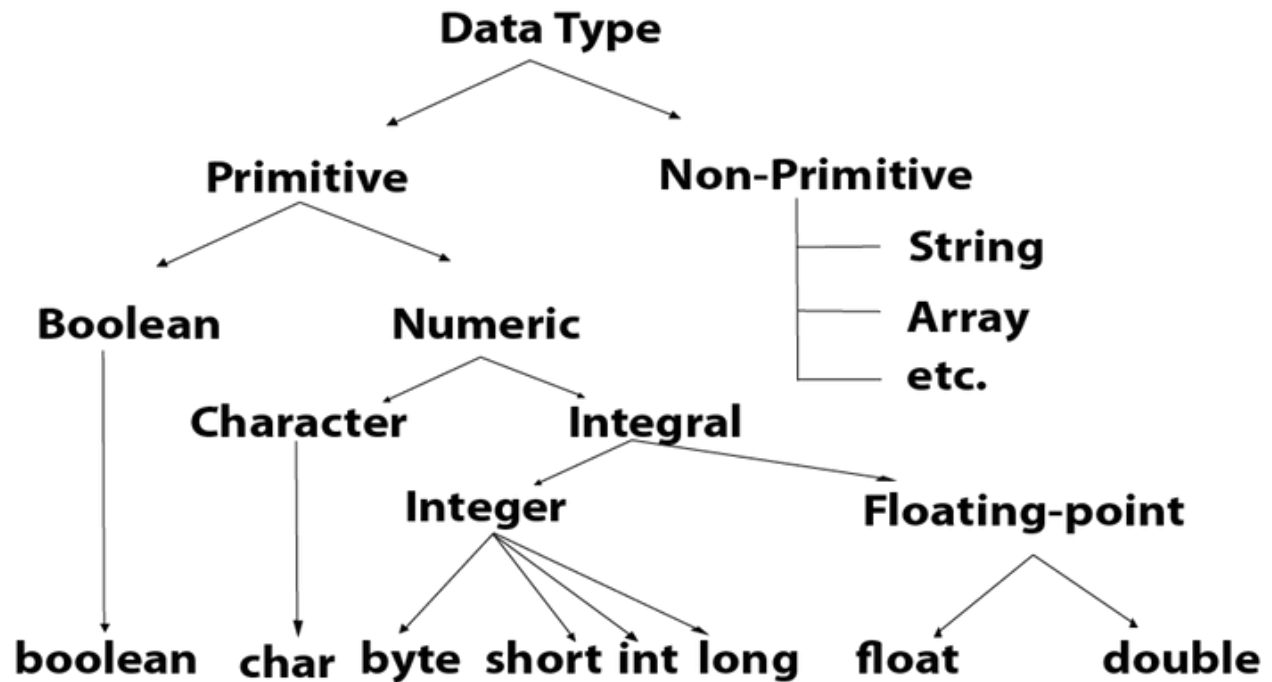
The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

Example: double d1 = 12.3

Char Data Type:

The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive). The char data type is used to store characters.

Example: char letterA = 'A'



Data Type	Default Value	Default size
boolean	False	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

Variable in Java:

- Variable is a memory location.
- is a container which holds the value while the **Java program** is executed. A variable is assigned with a data type.
- There are 3 types of variables:
 1. Local variables.
 2. Instance variables.
 3. Static Variables.

1. Local Variable:

- A variable declared inside the body of the method is called a local variable. You can use this variable only within that method.
- A local variable cannot be defined with the "**static**" keyword.

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.

3. Static variable:

A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of a static variable and share among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Example:

```
class A{  
    int data=50;//instance variable  
    static int m=100;//static variable  
    void method(){  
        int n=90;//local variable  
    }  
}
```

Constants

- **Constant** is a value that cannot be changed after assigning it.
 - Java does not directly support the constants.
 - Can define constants in Java by using the non-access modifiers static and final.

Syntax: `static final data_type identifier_name=value;`

`final data_type identifier_name=value;`

- The purpose of using the static modifier is to manage the memory.
- It also allows the variable to be available without loading any instance of the class in which it is defined.
- The final modifier represents that the value of the variable cannot be changed. It also makes the primitive data type immutable or unchangeable.

Type Conversion and Casting:

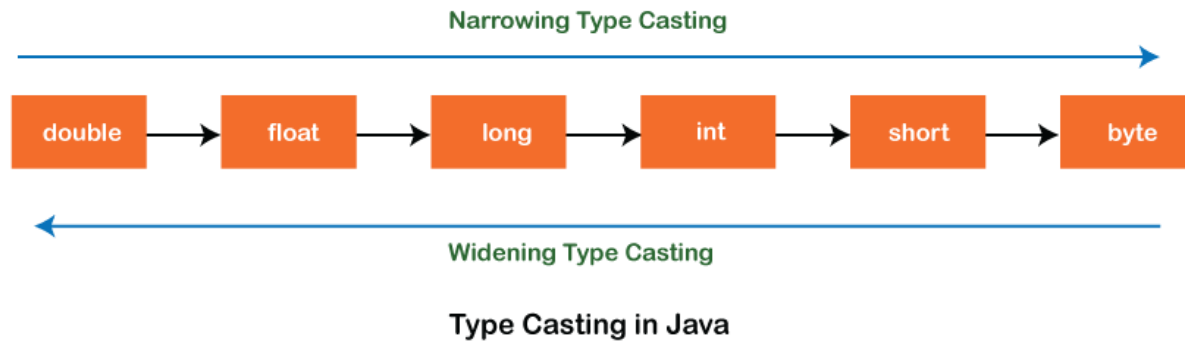
- It is a process that converts a data_type to another data_type both manually & automatically.
 - The automatic conversion is done by the compiler.
 - manual conversion performed by the programmer.
- There are two types type casting:
 1. Widening Conversion.
 2. Narrowing Conversion.
- **Narrowing Conversion:** Converts higher data type to lower data type.
 - Also known as explicit conversion or casting up.
 - This conversion is done by programmer.
- **double -> float -> long -> int -> char -> short -> byte**

Syntax: `dest_data_type var = (src_data_type) var`

Example:

`float x = 6;`

`int y=(int)x;`



Example Program for Widening:

```
public class WideningTypeCastingExample {  
    public static void main(String[] args) {  
        int x = 7;  
        long y = x;  
        System.out.println("Before conversion, int value "+x);  
        System.out.println("After conversion, float value "+z);  
    }  
}
```

Output:

Before conversion, the value is: 7
After conversion, the float value is: 7.0

```
class Test  
{  
    public static void main(String[] args)  
    {  
        int i = 100;  
  
        // automatic type conversion  
        long l = i;  
  
        // automatic type conversion  
        float f = l;  
        System.out.println("Int value "+i);  
        System.out.println("Long value "+l);  
        System.out.println("Float value "+f);  
    }  
}
```

Example:

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

Output:

```
10
10.0
```

Example Program for Narrowing:

```
public class WideningTypeCastingExample {
public static void main(String[] args) {
double d = 166.66;
long l = (long)d;
System.out.println("Before conversion: "+d);
System.out.println("After conversion into long type: "+l);
}
}
```

Output:

```
Before conversion: 166.66
After conversion into long type: 166
```

Example:

```
class Simple{
public static void main(String[] args){
float f=10.5f;
//int a=f;//Compile time error
int a=(int)f;
System.out.println(f);
System.out.println(a);
}}
```

Output:

```
10.5
10
```

Enumerations:

- An enumeration is user defined data type that contains named values. “**enum**” keyword is used to define enumerations.
- It is a list of constants.
- It is same as final variables.
- Introduced from JDK 1.5 version.

Syntax: enum enumeration_name{ identifier1, identifier2 ...}

EX: enum Car{FORD,TOYOTA,ROLLS_ROYCE}

- **Advantages:**

1. Increases type safety.
2. Easily applied to control flow statements & switch case statements.

- **Methods of Java enum:**

1. value() //returns an array containing all the values of the enum.
2. valueOf() //returns the value of given constant enum.
3. Ordinal() //returns the index of the enum value.

Example Program:

```
class EnumExample1{
public enum Season { WINTER, SPRING, SUMMER, FALL }
public static void main(String[] args) {
for (Season s : Season.values()){
System.out.println(s);
}
System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));
System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());
System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal()); }}
```

Output:

```
WINTER
SPRING
SUMMER
FALL
Value of WINTER is: WINTER
Index of WINTER is: 0
Index of SUMMER is: 2
```

```
class EnumExample1{ //defining enum within class
    public enum Season { WINTER, SPRING, SUMMER, FALL }
```



```
public static void main(String[] args) {  
    //printing all enum  
    for (Season s : Season.values()){  
        System.out.println(s);  
    }  
    System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));  
    System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());  
    System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());  
  
}}
```

Output:

```
WINTER  
SPRING  
SUMMER  
FALL  
Value of WINTER is: WINTER  
Index of WINTER is: 0  
Index of SUMMER is: 2
```

Note: Java compiler internally adds values(), valueOf() and ordinal() methods within the enum at compile time. It internally creates a static and final class for the enum.

values() method:

The Java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

valueOf() method:

The Java compiler internally adds the valueOf() method when it creates an enum. The valueOf() method returns the value of a given constant enum.

ordinal():

The Java compiler internally adds the ordinal() method when it creates an enum. The ordinal() method returns the index of the enum value.

Defining Java Enum:

The enum can be defined within or outside the class because it is similar to a class. The semicolon (;) at the end of the enum constants are optional.

Example:

1. **enum** Season { WINTER, SPRING, SUMMER, FALL }

Or,

1. **enum** Season { WINTER, SPRING, SUMMER, FALL; }

Both the definitions of Java enum are the same.

Java Enum Example:

Defined outside class

```
enum Season { WINTER, SPRING, SUMMER, FALL }
class EnumExample2{
public static void main(String[] args) {
Season s=Season.WINTER;
System.out.println(s);
}}
```

Output:

WINTER

Java Enum Example: Defined inside class

```
class EnumExample3{
enum Season { WINTER, SPRING, SUMMER, FALL; }//semicolon(;) is optional here
public static void main(String[] args) {
Season s=Season.WINTER;//enum type is required to access WINTER
System.out.println(s);
}}
```

Output:

WINTER

Java Enum Example: main method inside Enum

If you put the main() method inside the enum, you can run the enum directly.

```
enum Season {
WINTER, SPRING, SUMMER, FALL;
public static void main(String[] args) {
Season s=Season.WINTER;
System.out.println(s);
}
}
```

Output:

WINTER

Initializing specific values to the enum constants

- The enum constants have an initial value which starts from 0, 1, 2, 3, and so on.
- But, we can initialize the specific value to the enum constants by defining fields and constructors.
- Enum can have fields, constructors, and methods.

Example of specifying initial value to the enum constants

```
class EnumExample4{
enum Season{
WINTER(5), SPRING(10), SUMMER(15), FALL(20);

private int value;
private Season(int value){
this.value=value;
}
}
public static void main(String args[]){
for (Season s : Season.values())
System.out.println(s+" "+s.value);

}}
```

Output:

```
WINTER 5
SPRING 10
SUMMER 15
FALL 20
```

Constructor of the enum type is private. If you don't declare a private compiler, internally create a private constructor.

```
enum Season{
WINTER(10),SUMMER(20);
private int value;
Season(int value){
this.value=value;
}
}
```

Can we create an instance of Enum by a new keyword?

No, because it contains private constructors only.

Can we have an abstract method in the Enum?

Yes, Of course! we can have abstract methods and can provide the implementation of these methods.

Java Enum in a switch statement

```
class EnumExample5{
enum Day{ SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY}
public static void main(String args[]){
Day day=Day.MONDAY;

switch(day){
case SUNDAY:
System.out.println("sunday");
break;
case MONDAY:
System.out.println("monday");
break;
default:
System.out.println("other day");
}
}}
```

Output:

```
monday
```

Scope & lifetime of a variables

- Scope and lifetime are closely related but they are diff. Concepts.
- Scope of a variable is from point of declaration to end of block

Type of Variable	Scope	Lifetime
static	Entire program.	Until program terminates.
instance	Throughout the class except in static methods.	Until the object available in the memory.
local	Within the block where it is declared.	Until the control leaves the block where it is declared.
class	Throughout the class.	Until the end of program.

Local Variable

Local Variable is defined as a type of variable declared within a programming block . It can only be used inside the code block in which it is declared. The local variable exists until the block of the function is under execution. After that, it will be destroyed automatically.

Example of Local Variable

```
public int add(){
int a =4;
int b=5;
return a+b;
}
```

Here, 'a' and 'b' are local variables

Global Variable

A **Global Variable** in the program is a variable defined outside the method. It has a global scope means it holds its value throughout the lifetime of the program. Hence, it can be accessed throughout the program by any function defined within the program, unless it is shadowed.

Example:

```
int a =4;
int b=5;
public int add(){
return a+b;
}
```

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.

Static variable

A variable which is declared as static is called static variable. It cannot be local. You can create a single copy of a static variable and share among all the instances of the class. Memory allocation for static variables happens only once when the class is loaded in the memory.

Java Variable Example: Overflow

```
class Simple{
public static void main(String[] args){
//Overflow
int a=130;
byte b=(byte)a;
System.out.println(a);
System.out.println(b);
}}
```

Output:

130
-126

Java Variable Example: Adding Lower Type

```
class Simple{
public static void main(String[] args){
byte a=10;
byte b=10;
//byte c=a+b;//Compile Time Error: because a+b=20 will be int
byte c=(byte)(a+b);
System.out.println(c);
}}
```

Output: 20

Operators:

- It is a symbol used to perform operations on operands.

Operators in Java:

- Unary Operator.
- Arithmetic Operator.
- Shift Operator.
- Relational Operator.

- Bitwise Operator.
- Logical Operator.
- Ternary Operator.
- Assignment Operator.

Unary Operator

The Java unary operators require only one operand. Unary operators are used to perform various operations i.e.:

- incrementing/decrementing a value by one
- negating an expression
- inverting the value of a boolean

Unary Operator Example: ++ and --

```
class OperatorExample{
public static void main(String args[]){
    int x=10;
    System.out.println(x++);//10 (11)
    System.out.println(++x);//12
    System.out.println(x--);//12 (11)
    System.out.println(--x);//10
}}
```

Output:

```
10
12
12
10
```

Unary Operator Example 2: ++ and --

```
class OperatorExample{
public static void main(String args[]){
    int a=10;
    int b=10;
    System.out.println(a++ + ++a);//10+12=22
    System.out.println(b++ + b++);//10+11=21
}}
```

Output:

22

21

Unary Operator Example: ~ and !

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=-10;
boolean c=true;
boolean d=false;
System.out.println(~a);//-11 (minus of total positive value which starts from 0)
System.out.println(~b);//9 (positive of total minus, positive starts from 0)
System.out.println(!c);//false (opposite of boolean value)
System.out.println(!d);//true
}}
```

Output:

-11

9

false

true

Arithmetic Operators

Java arithmetic operators are used to perform addition(+), subtraction(-), multiplication(*), and division(/).

Arithmetic Operator Example

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

Output:

15

5

50

2

0

Arithmetic Operator Example: Expression

```
class OperatorExample{  
public static void main(String args[]){  
System.out.println(10*10/5+3-1*4/2);  
}}  

```

Output:

21

Left Shift Operator

The Java left shift operator << is used to shift all of the bits in a value to the left side of a specified number of times.

Left Shift Operator Example

```
class OperatorExample{  
public static void main(String args[]){  
System.out.println(10<<2);//10*2^2=10*4=40  
System.out.println(10<<3);//10*2^3=10*8=80  
System.out.println(20<<2);//20*2^2=20*4=80  
System.out.println(15<<4);//15*2^4=15*16=240  
}}  

```

Output:

40
80
80
240

Right Shift Operator

The Java right shift operator >> is used to move left operands value to right by the number of bits specified by the right operand.

Right Shift Operator Example

```
class OperatorExample{  
public static void main(String args[]){  
System.out.println(10>>2);//10/2^2=10/4=2  
System.out.println(20>>2);//20/2^2=20/4=5  
System.out.println(20>>3);//20/2^3=20/8=2  
}}  

```

Output:

2
5
2

Shift Operator Example: >> vs >>>

```
class OperatorExample{
public static void main(String args[]){
    //For positive number, >> and >>> works same
    System.out.println(20>>2);
    System.out.println(20>>>2);
    //For negative number, >>> changes parity bit (MSB) to 0
    System.out.println(-20>>2);
    System.out.println(-20>>>2);
}}
```

Output:

```
5
5
-5
1073741819
```

AND Operator Example: Logical && and Bitwise &

- The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.
- The bitwise & operator always checks both conditions whether the first condition is true or false.

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&& a<c); //false && true = false
System.out.println(a<b&a<c); //false & true = false
}}
```

Output:

```
false
false
```

AND Operator Example: Logical && vs Bitwise &

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b && a<c); //false && true = false
System.out.println(a); //10 because second condition is not checked
```

```
System.out.println(a<b&a++<c);//false && true = false
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
false
10
false
11
```

OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether the first condition is true or false.

```
class OperatorExample{
public static void main(String args[]){
    int a=10;
    int b=5;
    int c=20;
    System.out.println(a>b||a<c);//true || true = true
    System.out.println(a>b|a<c);//true | true = true
    //|| vs |
    System.out.println(a>b||a++<c);//true || true = true
    System.out.println(a);//10 because second condition is not checked
    System.out.println(a>b|a++<c);//true | true = true
    System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
true
true
true
10
true
11
```

Ternary Operator

Java Ternary operator is used as one linear replacement for **if-then-else** statement .It is the only conditional operator which takes three operands.

Ternary Operator Example

```
class OperatorExample{
public static void main(String args[]){
```

```
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

2

Another Example:

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

5

Assignment Operator

Java assignment operator is one of the most common operators. It is used to assign the value on its right to the operand on its left.

Assignment Operator Example

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

Output:

14

16

Assignment Operator Example

```
class OperatorExample{
public static void main(String[] args){
int a=10;
```

```

a+=3;//10+3
System.out.println(a);
a-=4;//13-4
System.out.println(a);
a*=2;//9*2
System.out.println(a);
a/=2;//18/2
System.out.println(a);
}}

```

Output:

```

13
9
18
9

```

Assignment Operator Example: Adding short

```

class OperatorExample{
public static void main(String args[]){
short a=10;
short b=10;
//a+=b;//a=a+b internally so fine
a=a+b;//Compile time error because 10+10=20 now int
System.out.println(a);
}}

```

Output:

Compile time error

After type cast:

```

class OperatorExample{
public static void main(String args[]){
short a=10;
short b=10;
a=(short)(a+b);//20 which is int now converted to short
System.out.println(a);
}}

```

Output:

20

Expression in Java:

A Java expression consists of variables, operators, literals, and method calls.

Example:

```
a=b+c*d-2;  
res=x+y-3.2;
```

Control Statements in Java:

Java provides 3 types of control flow statements:

- Decision Making statements.
- Loop statements.
- Jump statements.

Decision Making statements:

1. if statement.
2. if-else statement.
3. else-if statement.
4. Nested if-statement.
5. switch statement.

Simple if statement:

The general form of a simple if statement is

Syntax:

```
if (test expression){
```

```
statement-block;
```

```
}
```

```
statement-x;
```

If the test expression is true, the statement-block will be executed and followed by statement-x; otherwise statement-block will be skipped and the execution will jump to the statement-x.

Example:

```
if (basic > 5000){  
Comm= basic * 0.2;  
}
```

if..else statement:

The general form is

Syntax:

```
if (test expression){  
    //True stmt block  
}  
else{  
  
    //False stmt block  
}  
statement-x;
```

Write a Java program to find the greatest of two numbers.

```
class ifElse_Test{  
public static void main(String args[]){  
int a=10,b=5;  
if (a>b)  
{  
    System.out.println("a is big");}  
else{  
    System.out.println("b is big");  
}  
}
```

Nested if..else statement:

The general form is as follows:

Syntax:

```
if (condition-1){  
    if (condition-2){  
        statement(s);  
    } else {  
        statement(s);  
    } else  
{  
    if (condition-3)  
    {  
        statement(s);  
    }  
}
```

```
    }  
    else{  
        statement(s);  
    }  
  
}  
statement-x;
```

Write a Java program to find the greatest of three numbers.

```
class NestedIfElseTest{  
public static void main(String args[])  
{  
int a=15, b=10, c=5;  
if (a>b) {  
    if (a>c){  
        System.out.println("a is big");  
    }  
    else{  
        System.out.println("c is big");  
    }  
} else  
{  
    if(b>c){  
        System.out.println("b is big");  
    }  
    else{  
        System.out.println("c is big");  
    }  
}  
}
```


Else if ladder:

The general form is as follows:

Syntax:

```
if (condition-1) {  
    statement(s);  
}  
else if (condition-2)  
{  
    statement(s);  
}  
else if (condition-3)  
{  
    statement(s);  
}  
...  
else  
{  
    default-statement(s);  
}
```

Example:

```
if (marks > 69)  
    grade "Distinction";  
else if (marks 59)  
    grade "First";  
else if (marks > 49)  
    grade "second";  
else if (marks 39)  
    grade "third";  
else  
    grade = "fail";
```

switch statement:

The switch statement tests the values of a given variable (or expression) against a list of case values and when a match is found, a block of statements associated with that case is executed. The general form is as follows:

Syntax:

```
switch (expression){  
  case value-1:  
    statement-block-1;  
    break;  
  case value-2:  
    statement-block-2;  
    break;  
  .....  
  default:  
    default-block;  
} statement-x;
```

Write a Java program to find the addition, subtraction, multiplication, division depending upon the choice illustrating switch statement.

```
class SwitchTest{  
  public static void main(String args[])  
  {  
    int a = -40, b = -20, c, choice;  
    choice = 2;  
    switch(choice){  
      case 1:  
        c = a + b;  
        System.out.println(a + " + " + b + " = " + c);  
        break;  
      case 2:  
        c = a - b;  
        System.out.println(a + " - " + b + " = " + c);  
        break;  
      case 3:  
        c = a * b;  
        System.out.println(a + " * " + b + " = " + c);  
        break;  
      case 4:  
        c = a / b;  
        System.out.println(a + " / " + b + " = " + c);  
        break;  
      default :  
        System.out.println("Invalid Choice");  
    }  
  }  
}
```

Conditional operator statement:

This statement uses conditional operators. The conditional operators are ' ? ' and ' : '.

The general form is as follows:

[Variable =] (conditional expression)? expression1: expression2;

The conditional expression is evaluated first. If the result is true, the expression is evaluated and is returned as the value of the conditional expression. Otherwise, expression2 is evaluated and its value is returned.

Example:

```
if ( x<0)
    flag = 0;
else
    flag = 1;
```

Can be written as

```
flag (x<0) ? 0:1;
```

Write a Java program to find the greatest of two numbers illustrating conditional operator statements.

```
class CondOperatorTest
{
    public static void main(String args[])
    {
        int a 10, b=20, c;
        c = (a>b) ?a: b;
        System.out.println("Greatest Number = "+c);
    }
}
```

switch statement Fall Through:

It means it executes all statements after the first match if a break statement is not present.

Program:

```
public class SwitchExample {
    public static void main(String[] args) {
        int number=20;
        switch(number){
            case 10: System.out.println("10");
            case 20: System.out.println("20");
```

```
default: System.out.println("Not in 10, 20 or 30");
    }
}
}
```

switch statement with String:

- Java allows us to use strings in switch expressions.
- Java SE8 The case statement should be string literal.

Example Program:

```
public class SwitchExample {
public static void main(String[] args) {
    String levelString="Expert";
    int level=0;
    switch(levelString){
    case "Beginner": level=1;
    break;
    case "Expert": level=3;
    break;
    default: level=0;
    break; }
    System.out.println("Your Level is: "+level);
}}
```

Nested switch Statement: We can use switch statement inside other switch statements in java.

Syntax:

```
switch(n){
    case 1: // Nested switch
        switch(num) {
            case 10: statement 1;
            break;
            case 20: statement 2;
            break;
        }
    case 2: statement 2;
    break;
    case 3: statement 3;
```

```
break;  
default:  
}
```

enum in switch statement:

Java allows us to use enum in switch stmt.

Ex:

```
public class JavaSwitchEnumExample {  
    public enum Day { Sun, Mon, Tue }  
    public static void main(String args[])  
    {  
        Day[] DayNow = Day.values();  
        for (Day Now : DayNow)  
        {  
            switch (Now){  
  
                case Sun: System.out.println("Sunday");  
                    break;  
                case Mon: System.out.println("Monday")  
                    break;  
                case Tue: System.out.println("Tuesday");  
                    break;  
            }  
        }  
    }  
}
```

Java wrapper in switch statement:

Java allows us to use four wrapper classes: Byte, Short, Integer and Long in switch statement.

Ex:

```
public class WrapperInSwitchCaseExample {  
    public static void main(String args[]) {  
        Integer age = 18;  
        switch (age) {  
            case 16:  
                System.out.println("You are under 18.");  
                break;  
        }  
    }  
}
```

```
case (18): System.out.println("You are eligible for vote.");
           break;
case (65): System.out.println("You are senior citizen.");
           break;
default: System.out.println("Please give the valid age.");
         break;
    }
}
```

Loop statements:

1. for loop.
2. while loop. (entry control loop)
3. do-while loop. (exit control loop)
4. for-each loop.

for-each Loop :

for-each loop or enhanced for loop is introduced since J2SE 5.0

Syntax:

```
for(data_type variable : array | collection)
{
    //body of for-each loop
}
```

Ex:

```
class ForEachExample1 {
    public static void main(String args[]){
        int arr[]={ 12,13,14,44};
        for(int i:arr){
            System.out.println(i);
        }
    }
}
```

Loop	for	while	dowhile
Syntax	<pre>for(init;condition;incr/decr){ // code to be executed //statments }</pre>	<pre>while(condition){ //code to be executed }</pre>	<pre>do{ //code to be executed }while(condition);</pre>
Example	<pre>for(int i=1;i<=10;i++){ s.o.p(i); }</pre>	<pre>int i=1; while(i<=10){ s.o.p(i); i++; }</pre>	<pre>int i=1; do{ s.o.p(i); i++; }while(i<=10);</pre>

- 1. Initialization:** It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
- 2. Condition:** It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
- 3. Statement:** The statement of the loop is executed each time until the second condition is false.
- 4. Increment/Decrement:** It increments or decrements the variable value. It is an optional condition.

Write a Java Program to find the sum of individual digits.

Using a while loop.

```
import java.util.Scanner;
public class Digit_Sum
{
    public static void main(String args[])
    {
        int m, n, sum = 0;
        Scanner s = new Scanner(System.in);
        System.out.print("Enter the number:");
        m = s.nextInt();
        while(m > 0)
        {
            n = m % 10;
            sum = sum + n;
            m = m / 10;
        }
        System.out.println("Sum of Digits:"+sum);
    }
}
```

Using for loop:

```
class SumOfDigits
{
    public static void main(String arg[])
    {
        long n,s;
        Scanner sc=new Scanner(System.in);
        System.out.println("Enter a number ");
        n=sc.nextLong();
        s=sum(n);
        System.out.println("Sum of digits of a number is "+s);
    }
    static int sum(long num)
    {
        int sum=0;
        while(num!=0)
        {
```



```
        sum+=num%10;
        num/=10;
    }
    return sum;
}
```

Fibonacci series:

```
class FibonacciExample1{
public static void main(String args[])
{
    int n1=0,n2=1,n3,i,count=10;
    System.out.print(n1+" "+n2);//printing 0 and 1

    for(i=2;i<count;++i)//loop starts from 2 because 0 and 1 are already printed
    {
        n3=n1+n2;
        System.out.print(" "+n3);
        n1=n2;
        n2=n3;
    }
}
```

Prime Numbers:

```
public class PrimeExample{
public static void main(String args[]){
    int i,m=0,flag=0;
    int n=3;//it is the number to be checked
    m=n/2;
    if(n==0||n==1){
        System.out.println(n+" is not prime number");
    }else{
        for(i=2;i<=m;i++){
            if(n%i==0){
                System.out.println(n+" is not prime number");
                flag=1;
                break;
            }
        }
    }
}
```

```
    }  
    if(flag==0) { System.out.println(n+" is prime number"); }  
    }  
}
```

Palindrome Program

```
class PalindromeExample{  
    public static void main(String args[]){  
        int r,sum=0,temp;  
        int n=454;//It is the number variable to be checked for palindrome  
  
        temp=n;  
        while(n>0){  
            r=n%10; //getting remainder  
            sum=(sum*10)+r;  
            n=n/10;  
        }  
        if(temp==sum)  
            System.out.println("palindrome number ");  
        else  
            System.out.println("not palindrome");  
    }  
}
```

Reverse a number

```
public class ReverseNumberExample1  
{  
    public static void main(String[] args)  
    {  
        int number = 987654, reverse = 0;  
        while(number != 0)  
        {  
            int remainder = number % 10;  
            reverse = reverse * 10 + remainder;  
            number = number/10;  
        }  
        System.out.println("The reverse of the given number is: " + reverse);  
    }  
}
```

Java Program to Find Square Root of a Number

```
import java.util.Scanner;
public class FindSquareRootExample1
{
    public static void main(String[] args)
    {
        System.out.print("Enter a number: ");
        //creating object of the Scanner class
        Scanner sc = new Scanner(System.in);
        //reading a number form the user
        int n = sc.nextInt();
        //calling the method and prints the result
        System.out.println("The square root of " + n + " is: " + squareRoot(n));
    }
    //user-defined method that contains the logic to find the square root
    public static double squareRoot(int num)
    {
        //temporary variable
        double t;
        double sqrtroot=num/2;
        do
        {
            t=sqrtroot;
            sqrtroot=(t+(num/t))/2;
        }
        while((t-sqrtroot)!=0);
        return sqrtroot;
    }
}
```

Java Program to Find Largest of Three Numbers

```
import java.util.Scanner;
public class LargestNumberExample1
{
    public static void main(String[] args)
    {
        int a, b, c, largest, temp;
        //object of the Scanner class
        Scanner sc = new Scanner(System.in);
        //reading input from the user
        System.out.println("Enter the first number:");
        a = sc.nextInt();
```

```
System.out.println("Enter the second number:");
b = sc.nextInt();
System.out.println("Enter the third number:");
c = sc.nextInt();
//comparing a and b and storing the largest number in a temp variable
temp=a>b?a:b;
//comparing the temp variable with c and storing the result in the variable
largest=c>temp?c:temp;
//prints the largest number
System.out.println("The largest number is: "+largest);
}
}
```

Java Program to Display Even Numbers From 1 to 100

```
public class DisplayEvenNumbersExample1
{
    public static void main(String args[])
    {
        int number=100;
        System.out.print("List of even numbers from 1 to "+number+": ");
        for (int i=1; i<=number; i++)
        {
            //logic to check if the number is even or not
            //if i%2 is equal to zero, the number is even
            if (i%2==0)
            {
                System.out.print(i + " ");
            }
        }
    }
}
```

Java Program to Display Odd Numbers From 1 to 100

```
public class DisplayOddNumbersExample1
{
    public static void main(String args[])
    {
```

```
int number=100;
System.out.print("List of odd numbers from 1 to "+number+": ");
for (int i=1; i<=number; i++)
{
    //logic to check if the number is odd or not
    //if i%2 is not equal to zero, the number is odd
    if (i%2!=0)
    {
        System.out.print(i + " ");
    }
}
}
```

Java Program to Check if a Given Number is Perfect Square

```
import java.util.Scanner;
public class CheckPerfectSquareExample1
{
    //user-defined method that checks the number is perfect square or not
    static boolean checkPerfectSquare(double number)
    {
        //calculating the square root of the given number
        double sqrt=Math.sqrt(number);
        //finds the floor value of the square root and comparing it with zero
        return ((sqrt - Math.floor(sqrt)) == 0);
    }
    //main method
    public static void main(String[] args)
    {
        System.out.print("Enter any number: ");
        //object of the Scanner class
        Scanner sc=new Scanner(System.in);
        //reading a number of type double from the user
        double number=sc.nextDouble();
        //calling the user defined method
        if (checkPerfectSquare(number))
            System.out.print("Yes, the given number is perfect square.");
        else
            System.out.print("No, the given number is not perfect square.");
    }
}
```

```
}
}
```

Jump statements:

1. break.
2. continue.
3. return.

Stmt	break	continue	return
Syntax	break;	continue;	return value_to_be_returned;
Ex	<pre>for (int i = 0; i < 10; i++) { if (i == 4) { break; } s.o.p(i); }</pre>	<pre>for (int i = 0; i < 10; i++) { if (i == 4) { continue; } s.o.p(i); }</pre>	<pre>for (int i = 0; i < 10; i++) { if (i == 4) { return 23; } System.out.println(i); }</pre>

Command-line Arguments:

- Arguments are passed at the time of running the program. Input is given through command prompt.

Example:

```
class Xyz{  
    public static void main(String args[]){  
        System.out.println("Enter any number:" +args[0]);  
    }  
}
```

Compile: javac Xyz.java

run: java Xyz 23

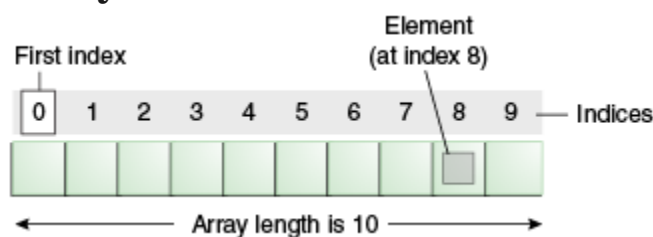
Example:

```
class Sum {  
    public static void main(String ar[]) {  
        int x,y,s;  
        x=Integer.parseInt(ar[0]);  
        y=Integer.parseInt(ar[1]);  
        s=x+y;  
        System.out.println("sum of " + x + " and " + y + " is " +s);  
    }  
}
```

Compile: javac Sum.java

run: java Sum 23 6

Arrays in Java:



Definition: It is a data structure where we store homogeneous elements as a single unit.

- Elements in array are stored in contiguous memory location.
- In Java array is an object.

Advantages:

1. Code optimization.

2. Random access

Disadvantages:

1. Size limit.

Syntax:

data_type array_name[size]=new [size];

Types of Arrays:

1. Single Dimensional Arrays.
2. Multi Dimensional Arrays.

Single Dimensional Arrays Syntax:

1. dataType[] arr; (or)
2. dataType []arr; (or)
3. dataType arr[];

Ex: int arr[]={1,2,3,6}; //declaration,instantiation,initialization

Multi Dimensional Arrays Syntax:

1. dataType[][] arr; (or)
2. dataType [][]arr; (or)
3. dataType arr[][]; (or)
4. dataType []arr[];

Syntax: data_type array_name[size][size]=new [size][size];

Ex: int a[][]=new int[3][3]; //instantiation

3D Dimensional Arrays Syntax:

dataType[][][] arr;

Syntax:data_type array_name[size][size][size]=new [size][size][size];

Ex: int a[][][]=new int[3][3][3]; //instantiation

Array_name.length : To know the size of an array, we use the “length” property.

Ex:

```
int a[]=new int[10];  
a.length; //size is 10
```


Jagged Arrays: It is an array of arrays such that member arrays can be of different sizes.

Syntax: data_type array_name[][] = new data_type[n][]; //n: no. of rows

array_name[] = new data_type[n1] //n1= no. of col in row-1

array_name[] = new data_type[n2] //n2= no. of col in row-2

.

.

array_name[] = new data_type[nk] //nk=no. of col in row-n

Ways to initialize Jagged Arrays :

1.

```
int arr_name[][] = new int[][] {  
    new int[] { 10, 20, 30 ,40},  
    new int[] { 50, 60, 70, 80, 90, 100},  
    new int[] { 110, 120}  
};  
OR
```
2.

```
int[][] arr_name = {  
    new int[] { 10, 20, 30 ,40},  
    new int[] { 50, 60, 70, 80, 90, 100},  
    new int[] { 110, 120}};
```
3.

```
int[][] arr_name = {  
    { 10, 20, 30 ,40},  
    { 50, 60, 70, 80, 90, 100},{110,120}};
```

Java Array

```
class Testarray{  
public static void main(String args[]){  
int a[]=new int[5];//declaration and instantiation  
a[0]=10;//initialization  
a[1]=20;  
a[2]=70;  
a[3]=40;  
a[4]=50;  
//traversing array  
for(int i=0;i<a.length;i++)//length is the property of array  
System.out.println(a[i]);  
}}
```

Output:

10
20
70
40
50

```
class Testarray1 {  
public static void main(String args[]){  
int a[]={33,3,4,5}; //declaration, instantiation and initialization  
//printing array  
for(int i=0;i<a.length;i++) //length is the property of array  
System.out.println(a[i]);  
}}
```

Output:

33
3
4
5

For-each Loop for Java Array

```
class Testarray1 {  
public static void main(String args[]){  
int arr[]={33,3,4,5};  
//printing array using for-each loop  
for(int i:arr)  
System.out.println(i);  
}}
```

Output:

33
3
4
5

Example of Multidimensional Java Array

```
class Testarray3 {  
public static void main(String args[]){  
//declaring and initializing 2D array
```

```

int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
    for(int j=0;j<3;j++){
        System.out.print(arr[i][j]+" ");
    }
    System.out.println();
}
}
}
}

```

Output:

```

1 2 3
2 4 5
4 4 5

```

Jagged Array in Java

```

class TestJaggedArray{
    public static void main(String[] args){
        //declaring a 2D array with odd columns
        int arr[][] = new int[3][];
        arr[0] = new int[3];
        arr[1] = new int[4];
        arr[2] = new int[2];
        //initializing a jagged array
        int count = 0;
        for (int i=0; i<arr.length; i++)
            for(int j=0; j<arr[i].length; j++)
                arr[i][j] = count++;

        //printing the data of a jagged array
        for (int i=0; i<arr.length; i++){
            for (int j=0; j<arr[i].length; j++){
                System.out.print(arr[i][j]+" ");
            }
            System.out.println();//new line
        }
    }
}

```

Output:

```

0 1 2
3 4 5 6
7 8

```

Addition of two matrices:

```
public class MatrixAdditionExample{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{2,4,3},{3,4,5}};
int b[][]={{1,3,4},{2,4,3},{1,2,4}};

//creating another matrix to store the sum of two matrices
int c[][]=new int[3][3]; //3 rows and 3 columns

//adding and printing addition of 2 matrices
for(int i=0;i<3;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j]; //use - for subtraction
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}
}}
```

Multiplication of matrices:

```
import java.util.Scanner;

class MatrixMultiplication
{
public static void main(String args[])
{
int m, n, p, q, sum = 0, c, d, k;

Scanner in = new Scanner(System.in);
System.out.println("Enter the number of rows and columns of first matrix");
m = in.nextInt();
n = in.nextInt();

int first[][] = new int[m][n];

System.out.println("Enter elements of first matrix");

for (c = 0; c < m; c++)
```

```
for (d = 0; d < n; d++)  
    first[c][d] = in.nextInt();
```

```
System.out.println("Enter the number of rows and columns of second matrix");  
p = in.nextInt();  
q = in.nextInt();
```

```
if (n != p)  
    System.out.println("The matrices can't be multiplied with each other.");  
else  
{  
    int second[][] = new int[p][q];  
    int multiply[][] = new int[m][q];
```

```
System.out.println("Enter elements of second matrix");
```

```
for (c = 0; c < p; c++)  
    for (d = 0; d < q; d++)  
        second[c][d] = in.nextInt();
```

```
for (c = 0; c < m; c++)  
{  
    for (d = 0; d < q; d++)  
    {  
        for (k = 0; k < p; k++)  
        {  
            sum = sum + first[c][k]*second[k][d];  
        }  
  
        multiply[c][d] = sum;  
        sum = 0;  
    }  
}
```

```
System.out.println("Product of the matrices:");
```

```
for (c = 0; c < m; c++)  
{  
    for (d = 0; d < q; d++)  
        System.out.print(multiply[c][d]+" ");
```

```
        System.out.print("\n");  
    }  
}  
}
```

UNIT II

Syllabus:

Introductions to Class and Objects: overview of classes, creations of objects, instant variables and methods, use of static, constructors, access control, usage of this, overloading methods and constructors, garbage collection.

Inheritance: overview, Super and Subclasses, Member access rules, types of Inheritance, super uses, method overriding, Dynamic method dispatch, abstract classes and methods, use of final, the Object class and its methods.

Overview of Classes:

Class: Class is a **blueprint** from which objects are created. It is a user-defined datatype.

- It represents the set of properties or methods that are common to all objects of one type.
- It describes the state and behaviour of a specific object.
- Class is a logical entity.
- It doesn't consume any space.

A class is declared by use of the class keyword. The classes that have been used up to this point are actually very limited examples of its complete form. Classes can (and usually do) get much more complex. A simplified general form of a class definition is shown here:

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



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

Eg: **public class** Car{

```
    int a;  
  
    void start();  
  
}
```

Syntax:

```
access_modifiers class <class_name>{  
  
    variable_declarations;  
  
    method_declarations;  
  
};
```

Example Program:

```
class Box {  
  
    double width;  
  
    double height;  
  
    double depth;  
  
}  
  
class BoxDemo2 {  
  
    public static void main(String args[]) {
```

```
Box mybox1 = new Box();

Box mybox2 = new Box();

double vol; // assign values to mybox1's instance variables

mybox1.width = 10;

mybox1.height = 20;

mybox1.depth = 15;

/* assign different values to mybox2's instance variables */

mybox2.width = 3;

mybox2.height = 6;

mybox2.depth = 9; // compute volume of first box

vol = mybox1.width * mybox1.height * mybox1.depth;

System.out.println("Volume is " + vol); // compute volume of second box

vol = mybox2.width * mybox2.height * mybox2.depth;

System.out.println("Volume is " + vol);

}

}
```

Output:

Volume is 3000.0

Volume is 162.0

Classes in Java Contains the Following:

- Fields.
- Methods.
- Constructors.
- Blocks.
- Nested classes and interfaces.

Example:

```
class Student {  
  
    int id; //field or data member or instance variable  
  
    void getData(int x){ //method declaration  
  
        x=id;  
  
    }  
  
    Student()  
  
        { //constructor  
  
            //stmts  
  
        }  
  
}
```

Creations of Objects:

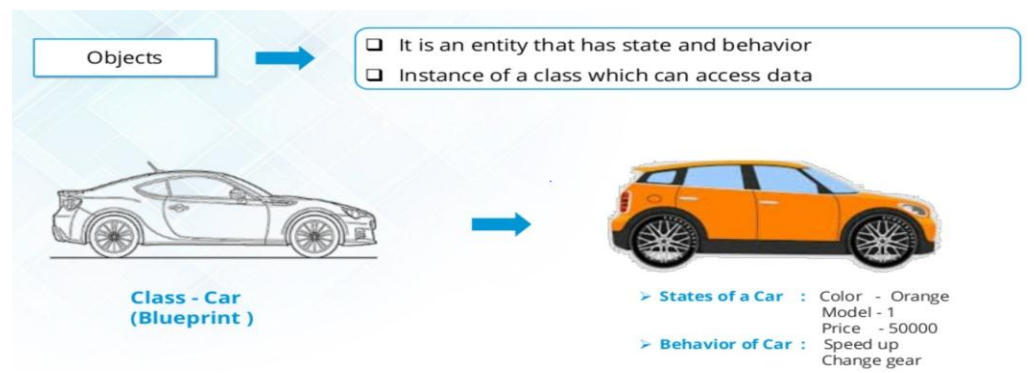
Object: An entity that has state and behaviour is called an object. It is an instance of class.

Or

An object is a real world and runtime entity.

Syntax: class_name object_name = **new** class_name();

Ex: Student s = new Student();



Obtaining objects of a class is a two-step process.

- First, you must declare a variable of the class type. This variable does not define an object. Instead, it is simply a variable that can refer to an object.
- Second, you must acquire an actual, physical copy of the object and assign it to that variable. You can do this using the new operator. The new operator dynamically allocates (that is, allocates at run time) memory for an object and returns a reference to it. This reference is, more or less, the address in memory of the object allocated by new. This reference is then stored in the variable. Thus, in Java, all class objects must be dynamically allocated.

Let's look at the details of this procedure. In the preceding sample programs, a line similar to the following is used to declare an object of type Box:

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

This statement combines the two steps just described. It can be rewritten like this to show each step more clearly:

```
Box mybox; // declare reference to object
```

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

Assigning Object Reference Variables

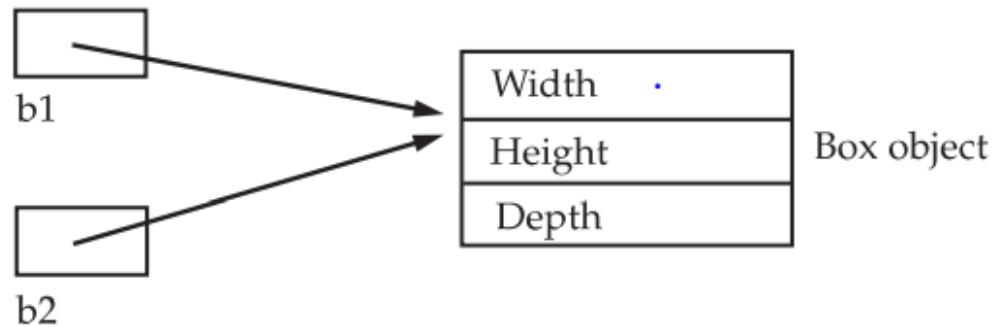
Object reference variables act differently than you might expect when an assignment takes place. For example, what do you think the following fragment does?

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

```
Box b2 = b1;
```

You might think that b2 is being assigned a reference to a copy of the object referred to by b1. That is, you might think that b1 and b2 refer to separate and distinct objects. However, this would be wrong. Instead, after this fragment executes, b1 and b2 will both refer to the same object. The assignment of b1 to b2 did not allocate any memory or copy any part of the original object. It simply makes b2 refer to the same object as does b1. Thus, any changes made to the object through b2 will affect the object to which b1 is referring, since they are the same object.

This situation is depicted here:



Although `b1` and `b2` both refer to the same object, they are not linked in any other way.

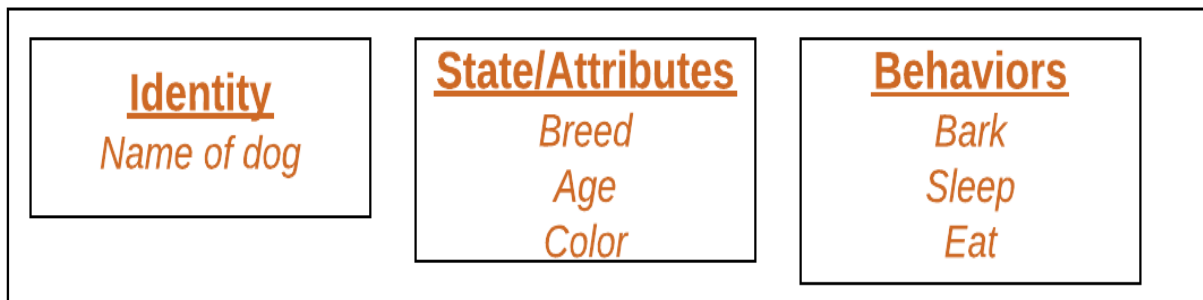
Note:

When you assign one object reference variable to another object reference variable, you are not creating a copy of the object, you are only making a copy of the reference

Characteristics of Object:

1. State.
2. Behaviour.
3. Identity.

Eg:



State: Represents the data(value) of an object.

Behaviour: Represents the functionality of an object such as deposit, withdraw, etc.

Identity: 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:

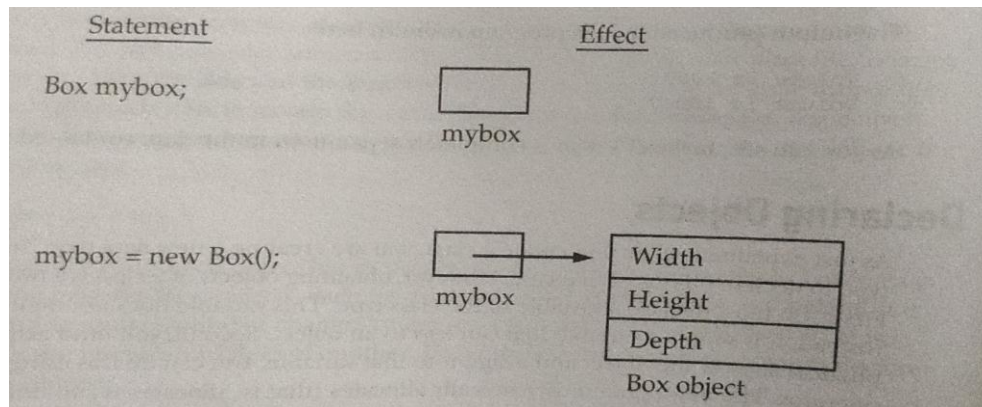
- An object is a real-world entity.
- An object is a runtime entity.
- The object is an entity which has state and behavior.
- The object is an instance of a class.

Steps to Create Object:

1. An object is created from class using a new keyword.
2. **Declaration:** A variable declaration with a variable name with an object type.
3. **Instantiation:** The 'new' keyword is used to create the object.
4. **Initialization:** The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

Ex:

```
public class Box{  
  
    public static void main(String args[]){  
  
        Box mybox = new Box();  
  
    }  
  
}
```



Accessing class members:

The instance variables and the methods of class are accessed through the objects.

Syntax:

`object_name.variable_name;`

`object_name.method_name(parameter_list)`

Example:

`mybox.a;`

`mybox.calculate();` //consider Box as class mybox as an object

Example Program:

```
class Student{  
  
    int id=6;  
  
    String name="Java";  
  
}
```



```
class TestStudent1  
  
{  
  
    public static void main(String args[])  
  
    {  
  
        Student s1=new Student();  
  
        System.out.println(s1.id);  
  
        System.out.println(s1.name);  
  
    }  
  
}
```

Different Ways to Create Object:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- Anonymous object etc

1. Test t = **new** Test() //using new keyword.

2. Using predefined method in java.lang package:

java.lang.Class.forName(String className)

Syntax:

Testo obj = (Test)Class.forName("<className>").newInstance();;

Parameters:

className:

This is the fully qualified name of the desired class.

Return Value:

This method returns the Class object for the class or interface with the specified name.

3. **Using clone():** It creates and returns a copy of the object.

```
Test t1 = new Test(); // creating object of class Test
```

```
Test t2 = (Test)t1.clone(); // creating clone of above object
```

4. **Deserialization:** De-serialization is the technique of reading an object from the saved state in a file. Refer Serialization/De-Serialization in java

```
FileInputStream file = new FileInputStream(filename);
```

```
ObjectInputStream in = new ObjectInputStream(file);
```

```
Object obj = in.readObject();
```

5. Anonymous Objects:

Anonymous objects are the objects that are instantiated but are not stored in a reference variable.

Anonymous simply means nameless. It can be used at the time of object creation only.

Ex:

```
public class Calculation{  
  
    void fact(int n){  
  
        int fact=1;  
  
        for(int i=1;i<=n;i++){
```

```
        fact=fact*i;

    }

    System.out.println("factorial is "+fact); }

    public static void main(String args[]){

        new Calculation().fact(5);//calling method with anonymous object

    } }
```

Creating multiple objects by one type only

- We can create multiple objects by one type only as we do in case of primitives.
 - `int a=10, b=20;` //Initialization of primitive variables:
- Initialization of reference variables:
 - `Rectangle r1=new Rectangle(),r2=new Rectangle();`//creating two objects

Ex:

```
class Rectangle{

    int length;

    int width;

    void insert(int l,int w){

        length=l;

        width=w;

    }

    void calculateArea(){System.out.println(length*width);}

}
```

```
class TestRectangle2{  
  
    public static void main(String args[]){  
  
        Rectangle r1=new Rectangle(),r2=new Rectangle();  
  
        r1.insert(11,5);  
  
        r2.insert(3,15);  
  
        r1.calculateArea();  
  
        r2.calculateArea();  
  
    }  
  
}
```

Different ways to initialize object:

Initializing an object means storing data into the object.

There are 3 ways to initialize:

1. By reference variable
2. By method
3. By constructor

By Reference Variable:

1. declaring the **reference variable**;
2. **using** the new operator to build an **object and create a reference** to the **object**;
and.
3. storing the **reference in** the **variable**.

Ex:

```
class Student

{

    int id;

    String name;

}

class TestStudent2

{

    public static void main(String args[])

    {

        Student s1=new Student();

        s1.id=101;

        s1.name="JAVA";

        System.out.println(s1.id+" "+s1.name);//printing members with a white space

    }

}
```

Output:

```
101 JAVA
```

We can also create multiple objects and store information in it through reference variables.

```
class Student
```

```
{  
  
    int id;  
  
    String name;  
  
}  
  
class TestStudent3{  
  
    public static void main(String args[]){  
  
        //Creating objects  
  
        Student s1=new Student();  
  
        Student s2=new Student();  
  
        //Initializing objects  
  
        s1.id=101;  
  
        s1.name="C++";  
  
        s2.id=102;  
  
        s2.name="JAVA";  
  
        //Printing data  
  
        System.out.println(s1.id+" "+s1.name);  
  
        System.out.println(s2.id+" "+s2.name);  
  
    }  
  
}
```

Output:

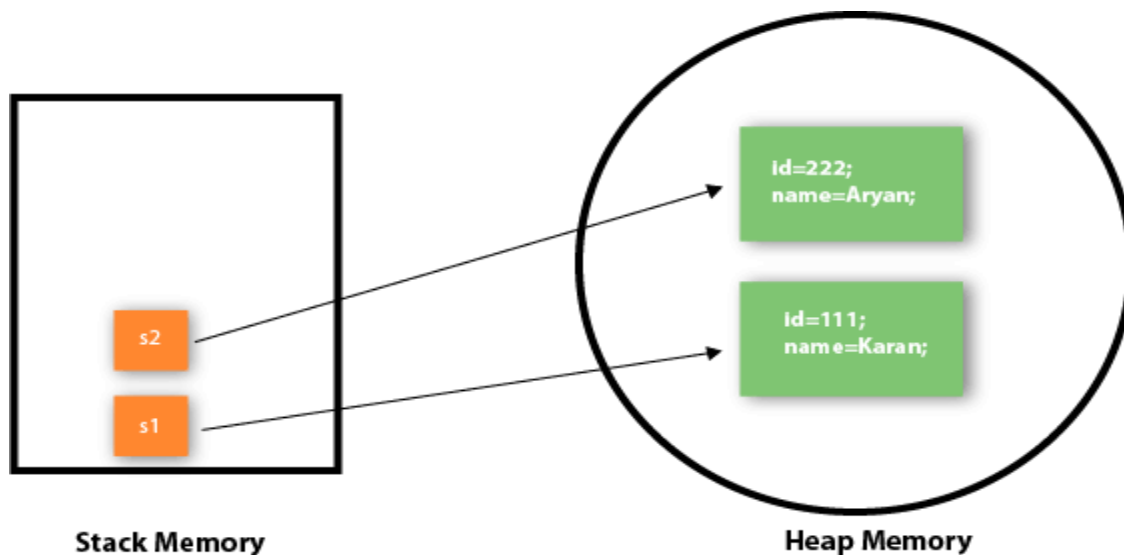
101 C++

102 JAVA

By Method:

1. Object gets the memory in the heap memory area.
2. The reference variable refers to the object allocated in the heap memory area.

Ex:



As you can see in the above figure, the object gets the memory in the heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

Example Program:

```
class Student{  
  
    int rollno;  
  
    String name;  
  
    void insertRecord(int r, String n){
```

```
        rollno=r;

        name=n;

    }

    void displayInformation(){

        System.out.println(rollno+" "+name);}

    }

    class TestStudent4{

        public static void main(String args[]){

            Student s1=new Student();

            Student s2=new Student();

            s1.insertRecord(111,"JAVA");

            s2.insertRecord(222,"C++");

            s1.displayInformation();

            s2.displayInformation();

        }

    }
```


Instance Variables & Methods:

Instance Variable: A variable declared inside the class but outside the body of the method.

- It is not declared as static.
- It is called instance variable because its value is instance specific and is not shared among instances.

Ex:

```
class A{  
  
    int data=50;//instance variable  
  
    static int m=100;//static variable  
  
    void method(){  
  
        int n=90;//local variable  
  
    } }
```

Example:

```
class Person{  
  
    String name;  
  
    int age;  
  
    void talk(){  
  
        System.out.println("Hello" +name);  
  
        System.out.println("My age is " +age);  
  
    } }
```

```
public static void main(String args[]){  
  
    Person a = new Person();  
  
    //initializing instance variable using reference  
  
    a.name="Xyz";  
  
    a.age=22;  
  
    a.talk();  
  
    }  
  
}
```

Methods:

A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation.

Syntax:

```
access_specifier  return_type  method_name(parameter_list)  
  
{  
  
    -----  
  
    -----  \\method_body  
  
    -----  
  
}
```

Ex:

```
public int add(int a,int b)

{

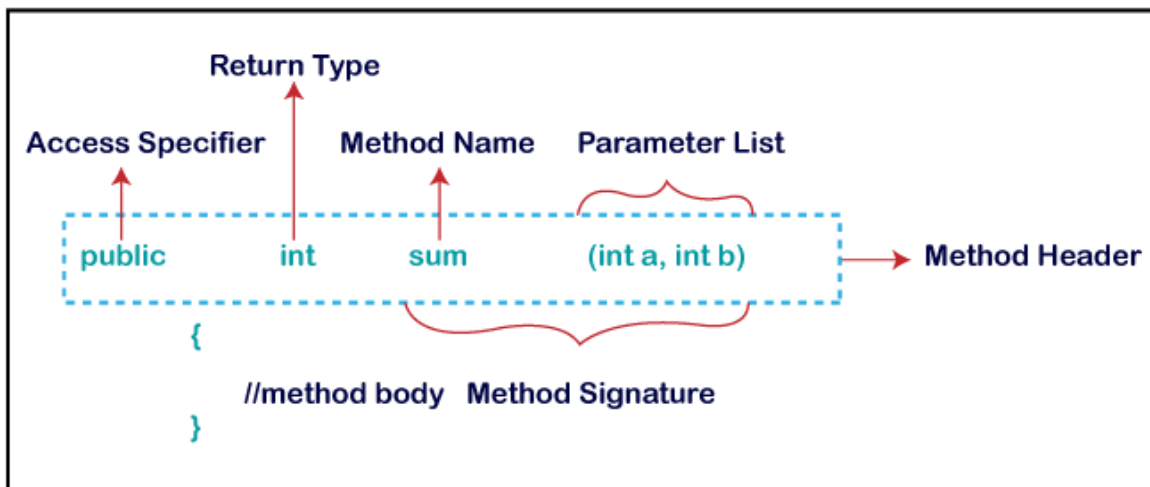
    return (a+b);

}
```

Method Signature: Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.

Ex: add(int a,int b) is a method signature in above example.

Method Declaration



Return Type: Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use the void keyword.

Method Name: It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction()**. A method is invoked by its

name.

Parameter List: It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.

Method Body: It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming conventions for writing a Method:

Single-word method name: sum(), area()

Multi-word method name: areaOfCircle(), stringComparision()

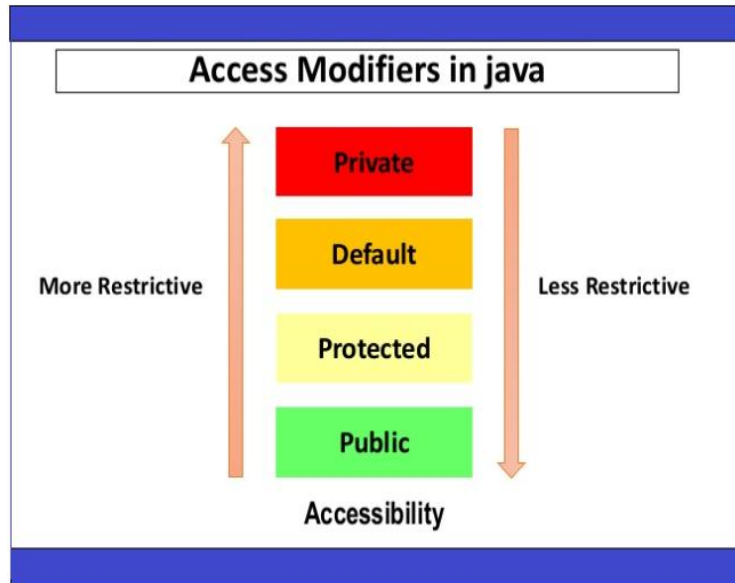
It is also possible that a method has the same name as another method name in the same class, it is known as **method overloading**.

Access Specifiers:

- Access specifier or modifier is the access type of the method. It specifies the visibility of the method.

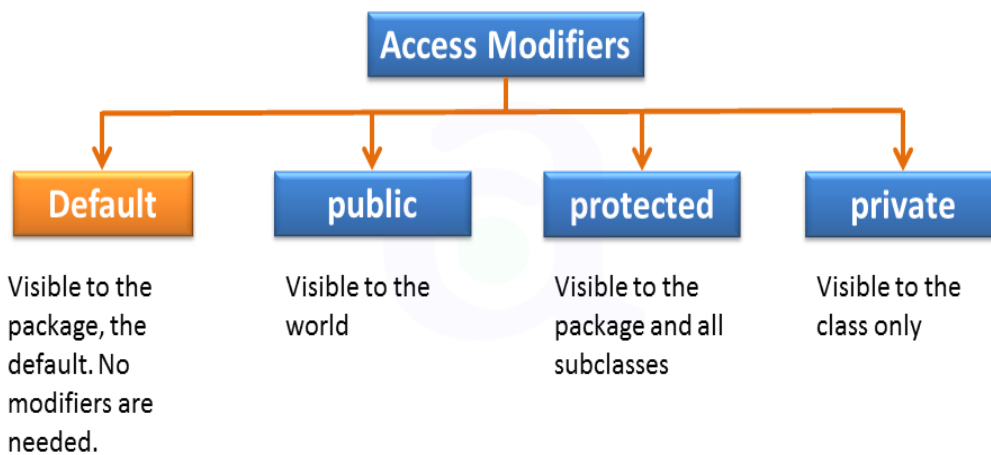
There are 4 access specifiers:

1. public.
2. private.
3. protected.
4. default.



- **Public:** The method is accessible by all classes when we use a public specifier in our application.
- **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
- **Protected:** When we use protected access specifiers, the method is accessible within the same package or subclasses in a different package.
- **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

Access Modifiers	Non-Access Modifiers
private default or No Modifier protected public	static final abstract synchronized transient volatile strictfp



Types of Methods:

There are 2 types of methods:

1. Predefined methods.
2. User-defined methods.

Predefined / Standard Library / built-in methods:

- Methods that are already defined in java class libraries.
- Each and every predefined method is defined inside a class. Such as **print()** method is defined in the **java.io.PrintStream** class. It prints the statement that we write inside the method. For example, **print("Java")**, it prints Java on the console.

Ex: **length(), equals(), compareTo(), sqrt(), etc.**

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        // using the max() method of Math class  
  
        System.out.print("The maximum number is: " + Math.max(9,7));  
  
    }  
  
}
```

Output:

The maximum number is: 9

User-Defined Methods:

- The method written by a user or programmer is called a user-defined method.

Ex:

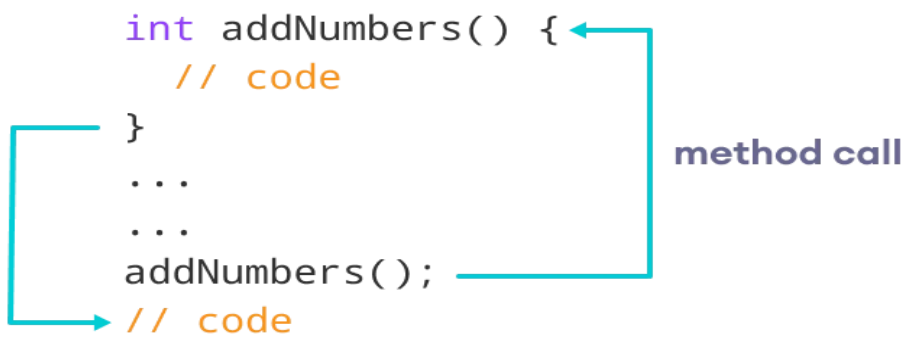
```
public static void findEvenOdd(int num) //user defined method  
  
{  
  
    //method body  
  
    if(num%2==0)  
  
        System.out.println(num+" is even");  
  
}
```

else

```
System.out.println(num+" is odd");  
  
}
```

Calling or invoking a user-defined method:

When we call a user defined method the control transfers to the called method.



Example Program:

```
import java.util.Scanner;  
  
public class EvenOdd {  
  
    public static void main (String args[]) {  
  
        Scanner scan=new Scanner(System.in);  
  
        System.out.print("Enter the number: ");  
  
        int num=scan.nextInt();  
  
        findEvenOdd(num); //method calling  
  
    }  
}
```



```
public static void findEvenOdd(int num) //user defined method{  
  
    if(num%2==0)  
  
        System.out.println(num+" is even");  
  
    else  
  
        System.out.println(num+" is odd"); }  
  
}
```

Actual and Formal Parameter:

- When a parameter is passed into a method, it is called **argument**.
- The **actual parameter** is the argument which is used in the method call whereas the **formal parameter** is the argument which is used in the method definition.

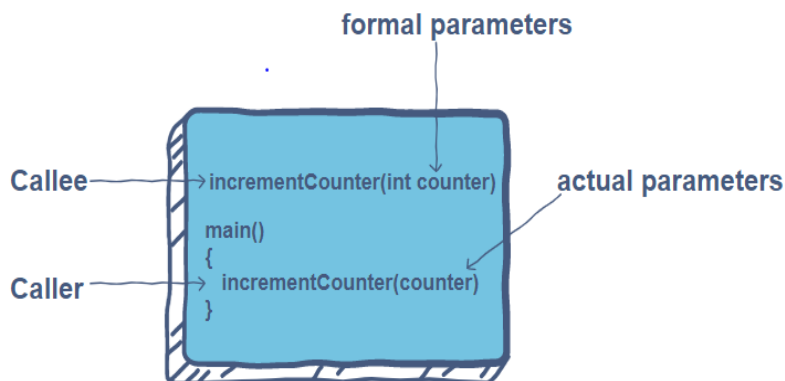
Ex:

```
public class Addition {  
  
    public static void main(String[] args) {  
  
        int a = 19;  
  
        int b = 5;  
  
        int c = add(a, b); //a and b are actual parameters ,method calling  
  
        System.out.println("The sum of a and b is= " + c);  
  
    }  
  
    //user defined method  
  
    public static int add(int n1, int n2) //n1 and n2 are formal parameters
```

```
{  
  
int s;  
  
s=n1+n2;  
  
return s; //returning the sum  
  
}  
  
}
```

Output:

The sum of a and b is= 24



-
- **Pass by Value:** It is a process in which the method parameter values are copied to another variable and instead this object copied is passed. This is known as call by Value.

Ex:

```
import java.io.*;  
  
public class Swap {  
  
    static void swap(int a, int b)  
  
    {
```

```
int temp = a;

a = b;

b = temp;

}
```

Pass by Reference:

It means to pass the reference of an argument in the calling method to the corresponding formal parameter of the called method so that a copy of the address of the actual parameter .

Pass by Value vs Pass by Reference:

Parameters	Pass by value	Pass by reference
Arguments	In this method, a copy of the variable is passed.	In this method, a variable itself is passed.
Effect	Change in the variable also doesn't affect the value of the variable outside the method.	Change in the variable also affects the value of the variable outside the method.
Alteration of value	Does not allow you to make any changes in the actual variables.	Allows you to make changes in the values of variables by using method calls.
Value modification	Original value not modified.	The original value is modified.
Memory Location	Actual and formal arguments will be created in different memory location	Actual and formal arguments will be created in the same memory location

Safety	Actual arguments remain safe as they cannot be modified accidentally.	Actual arguments are not Safe. They can be accidentally modified.
---------------	---	---

Instance & Static methods in java:

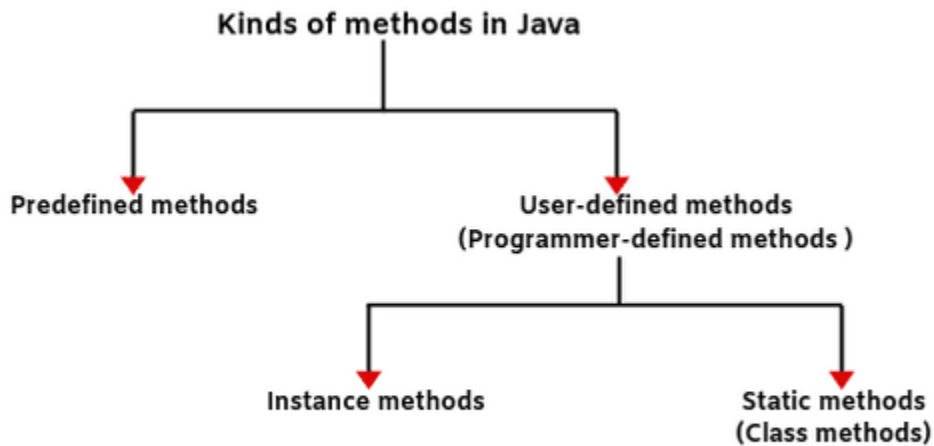


Fig: Basic kinds of methods in Java

Instance Method:

- The method of the class is known as an **instance method**.
- It is a **non-static** method.
- Before calling or invoking the instance method, it is necessary to create an object of its class.

There are two types of instance method:

- **Accessor Method**
- **Mutator Method**

Accessor Method: The method(s) that reads the instance variable(s) is known as the accessor method.

- It is prefixed with the word **get**.
- It is also known as **getters**.
- It returns the value of the private field.
- It is used to get the value of the private field.

Example

```
public int getId()

{

    return Id;

}
```

Mutator Method: The method(s) read the instance variable(s) and also modify the values.

- It is prefixed with the word **set**. It is also known as **setters** or **modifiers**.
- It does not return anything.
- It accepts a parameter of the same data type that depends on the field.
- It is used to set the value of the private field.

Example

```
public void setRoll(int roll)

{

    this.roll = roll;

}
```

Abstract Method:

- The method that does not have a method body is known as an abstract method. (or)
- In other words, without an implementation is known as an abstract method.
- It always declares in the **abstract class**.
- To create an abstract method, we use the keyword **abstract**.

Syntax

abstract void method_name();

Ex:

abstract void add();

Factory method:

- It is a method that returns an object to the class to which it belongs.
- All static methods are factory methods.

Ex:

NumberFormat obj = NumberFormat.getNumberInstance();

Use of static:

- The static keyword is mainly used for memory management.

Static can be applied to:

1. Variable
2. Method
3. Block
4. Nested class

static variables can be used without having an instance of class.

Syntax: **static** data_type variable;

- A single copy to be shared by all instances of the class.
- A static variable can be accessed directly by the class name and doesn't need any object.
- The static variable gets memory only once in the class area at the time of class loading.

Ex: **static int** age;

Example without static:

```
class Counter{

    int count=0;

    Counter(){

        count++; //incrementing value

        System.out.println(count);

    }

    public static void main(String args[]){

        Counter c1=new Counter();

        Counter c2=new Counter();

        Counter c3=new Counter();

    }

}
```

Output:

1

1

1

Example with static:

```
class Counter2{

    static int count=0;//will get memory only once and retain its value

    Counter2(){

        count++; //incrementing the value of static variable

        System.out.println(count);

    }

    public static void main(String args[]){

        Counter2 c1=new Counter2();

        Counter2 c2=new Counter2();

        Counter2 c3=new Counter2();

    }

}
```

Output:

1

2

3

Java static method:

- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access a static data member and can change the value of it.

Ex:

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

Ex:

```
public class Display {  
  
    public static void main(String[] args) {  
  
        show();  
  
    }  
  
    static void show() {  
  
        System.out.println("It is an example of static method.");  
  
    }  
  
}
```

Output:

It is an example of a static method.

Restrictions for the static method:

There are two main restrictions for the static method. They are:

1. They can only call other static methods.
2. They must only access static data.
3. They cannot refer to this or super in any way.

Why is the Java main method static?

- It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then calls main() method that will lead to the problem of extra memory allocation.

Static block

- Is used to initialize the static data member.
- It is executed before the main method at the time of classloading.

Ex:

```
class A2{  
  
    static{System.out.println("static block is invoked");}  
  
    public static void main(String args[]){  
  
        System.out.println("Hello main");  
  
    }  
  
}
```

Output:

static block is invoked

Hello main

Can we execute a program without the main() method?

- No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the main method.

Ex:

```
class A3{  
  
    static{  
  
        System.out.println("static block is invoked");  
  
        System.exit(0);  
  
    }  
  
}
```

Output:

static block is invoked

// Demonstrate static variables, methods, and blocks.

```
class UseStatic {  
  
    static int a = 3;  
  
    static int b;  
  
    static void meth(int x) {  
  
        System.out.println("x = " + x);  
  
        System.out.println("a = " + a);  
  
    }  
  
}
```

```
        System.out.println("b = " + b);
    }

    static {

        System.out.println("Static block initialized.");

        b = a * 4;

    }

    public static void main(String args[]) {

        meth(42);

    }

}
```

Explanation:

As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes, which prints a message and then initializes b to a*4 or 12. Then main() is called, which calls meth(), passing 42 to x. The three println() statements refer to the two static variables a and b, as well as to the local variable x.

Here is the output of the program:

Static block initialized.

x = 42

a = 3

b = 12

Outside of the class in which they are defined, static methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by

the dot operator.

For example, if you wish to call a static method from outside its class, you can do so using the following general form:

classname.method()

Here, classname is the name of the class in which the static method is declared. This format is similar to that used to call non-static methods through object-reference variables.

A static variable can be accessed in the same way—by use of the dot operator on the name of the class. This is how Java implements a controlled version of global methods and global variables. Here is an example. Inside main(), the static method callme() and the static variable b are accessed through their class name StaticDemo.

```
class StaticDemo {  
  
    static int a = 42;  
  
    static int b = 99;  
  
    static void callme() {  
  
        System.out.println("a = " + a);  
  
    }  
  
}  
  
class StaticByName {  
  
    public static void main(String args[]) {  
  
        StaticDemo.callme();  
  
        System.out.println("b = " + StaticDemo.b);  
  
    }  
  
}
```

```
}
```

Output:

a = 42

b = 99

Constructors

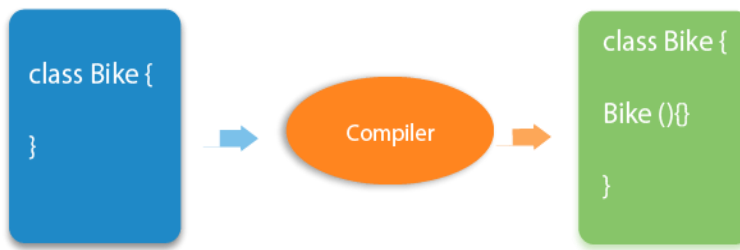
A constructor is a special method that is used to initialize an object.

- It is called when an instance of the **class** is created.
- At the time of calling the constructor, memory for the object is allocated in the memory.
- Every time an object is created using the `new()` keyword, at least one constructor is called.

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

Rules for creating Java constructor:

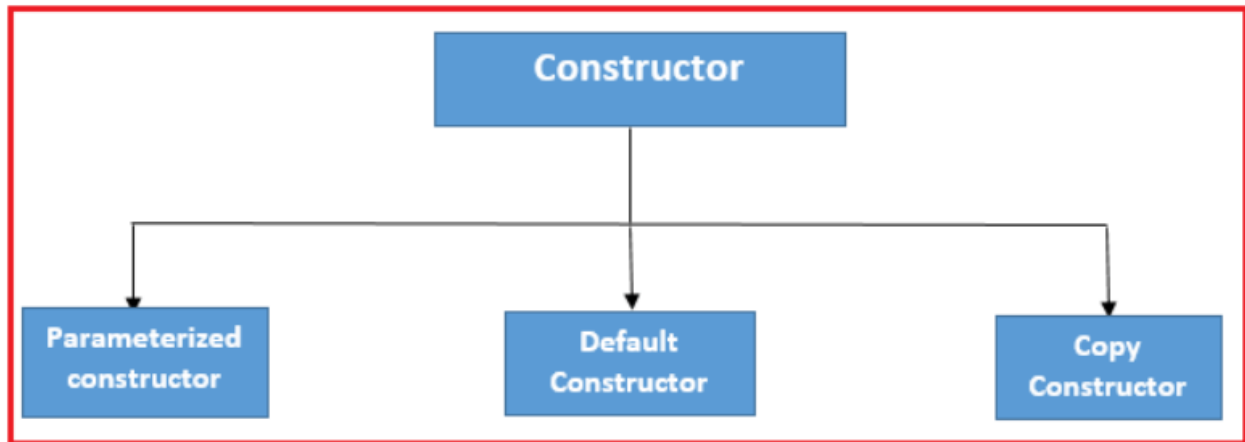
1. Constructor name must be the same as its class name.
2. A Constructor must have no explicit return type.
3. A Java constructor cannot be abstract, static, final, and synchronized.
4. We can use access modifiers while declaring a constructor.
5. If there is no constructor, the compiler creates a default constructor.



Constructor vs method:

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be the same as the class name.	The method name may or may not be the same as the class name.

Types of constructors:



Default Constructor:

- A default constructor is a **0 argument constructor**.
- To assign default values to the newly created objects is the main responsibility of the default constructor.
- The access modifier of default constructor is always the same as a class modifier but this rule is applicable only for “**public**” and “**default**” modifiers.

Syntax:

```
class Test {  
  
    Test() {  
  
        // constructor body  
  
    }  
  
}
```


Example:

```
class Main {  
  
    private String name;  
  
    Main() {  
  
        System.out.println("Constructor Called:");  
  
        name = "Xyz";  
  
    }  
  
    public static void main(String[] args) {  
  
        Main obj = new Main();  
  
        System.out.println("The name is " + obj.name);  
  
    }  
  
}
```

Parameterized Constructors:

- The parameterized constructors are the constructors having a **specific number of arguments** to be passed.
- A parameterized constructor is written explicitly by a programmer.
- A Java constructor can also accept one or more parameters. Such constructors are known as parameterized constructors

Syntax:

```
class Test {  
  
    Test(parameters) {  
  
        // constructor body  
  
    }  
  
}
```

```
}
```

```
}
```

Example:

```
class Main {
```

```
String languages;
```

```
    Main(String lang) { //parameterized constructor
```

```
        languages = lang;
```

```
        System.out.println(languages + " Programming Language");
```

```
}
```

```
public static void main(String[] args) {
```

```
    Main obj1 = new Main("Java");
```

```
    Main obj2 = new Main("Python");
```

```
    Main obj3 = new Main("C");
```

```
}
```

```
}
```

Copy Constructor:

- A **copy constructor** in a **Java** class is a **constructor** that creates an object using another object of the same **Java** class.

Use of Copy Constructor:

- Create a copy of an object that has multiple fields.
- Avoid the use of the `Object.clone()` method.

Advantages of Copy Constructor

- If a field declared as final, the copy constructor can change it.
- There is no need for typecasting.
- Its use is easier if an object has several fields.

Creating a Copy Constructor

- Create a constructor that accepts an object of the same class as a parameter.
- Copy each field (variable) object into the newly created instance.

Syntax:

```
public class Fruits {  
  
    private double price;  
  
    private String name;  
  
    public Fruits(Fruits fruits) //copy constructor  
  
    {  
  
        //getters  
  
    }  
  
}
```

Default Constructor vs Parameterized Constructor:

Default constructor	Parameterized constructor
A constructor which takes no arguments is known as the default constructor.	A constructor which takes one or more arguments is known as parameterized constructor.

The compiler inserts a default no-arg constructor after compilation if there is no explicit constructor defined in a class.	When the parameterized constructor is defined in a class, then the programmer needs to define the default no-arg constructor explicitly if required.
No need to pass any parameters while constructing new objects using the default constructor.	At least one or more parameters need to be passed while constructing new objects using argument constructors.
A default constructor is used for initializing objects with the same data	Whereas parameterized constructor is used to create distinct objects with different data

Example Programs:

/ Here, Box uses a constructor to initialize the dimensions of a box. */*

```
class Box {  
  
    double width;  
  
    double height;  
  
    double depth; // This is the constructor for Box.  
  
    Box() {  
  
        System.out.println("Constructing Box");  
  
        width = 10;  
  
        height = 10;  
  
        depth = 10;  
    }  
}
```

```
    } // compute and return volume

    double volume() {

        return width * height * depth;

    }

}

class BoxDemo6 {

    public static void main(String args[]) {

        // declare, allocate, and initialize Box objects

        Box mybox1 = new Box();

        Box mybox2 = new Box();

        double vol;

        // get volume of first box

        vol = mybox1.volume();

        System.out.println("Volume is " + vol);

        // get volume of second box

        vol = mybox2.volume();

        System.out.println("Volume is " + vol);

    }

}
```

Output:

Constructing Box

Constructing Box

Volume is 1000.0

Volume is 1000.0

Another Example:

/ Here, Box uses a parameterized constructor to initialize the dimensions of a box. */*

```
class Box {  
  
    double width;  
  
    double height;  
  
    double depth;  
  
    // This is the constructor for Box.  
  
    Box(double w, double h, double d) {  
  
        width = w;  
  
        height = h;  
  
        depth = d;  
  
    } // compute and return volume  
  
    double volume() {  
  
        return width * height * depth;  
  
    }  
  
}  
  
class BoxDemo7 {  
  
    public static void main(String args[]) {
```

```
// declare, allocate, and initialize Box objects

Box mybox1 = new Box(10, 20, 15);

Box mybox2 = new Box(3, 6, 9);

double vol;

// get volume of first box

vol = mybox1.volume();

System.out.println("Volume is " + vol);

// get volume of second box

vol = mybox2.volume();

System.out.println("Volume is " + vol);

}

}
```

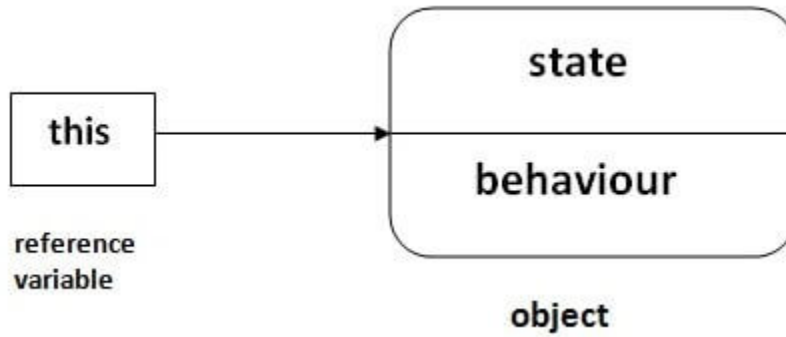
Output:

Volume is 3000.0

Volume is 162.0

Use of this keyword:

There can be a lot of usage of java **this keyword**. In java, this is a **reference variable** that refers to the current object.



Basically, “this” keyword can be used in two ways.

1. **this.**
2. **this()**

1. **this.**

It can be used to differentiate variables of the class and formal parameters of method or constructor. Not only that, it always points to the current class object. Syntax of this keyword is as shown below:

Syntax:

this.data member of the current class

Note: If there is any variable which is preceded by “this”, then JVM treats that variable as a class variable.

2. **this()**

It can be used to call one constructor within another without creating the objects multiple times for the same class.

Syntax:

- 1 `this();` // call no parameterized or default constructor
- 2 `this(value1,value2,.....)` //call parameterized constructor

Usage of java this keyword

1. this can be used to refer to the current class instance variable. `//this.variable_name;`
2. this can be used to invoke the current class method (implicitly). `//this.method_name();`
3. this() can be used to invoke the current class constructor. `//this();`
4. this can be passed as an argument in the method call.

```
class S2{  
  
    void m(S2 obj){  
  
        System.out.println("method is invoked");  
  
    }  
  
    void p(){  
  
        m(this);  
  
    }  
}
```

5. this can be passed as an argument in the constructor call.

```
class B{  
  
    A4 obj;  
  
    B(A4 obj){  
  
    }  
}
```

```
        this.obj=obj;

    }

    void display(){

        System.out.println(obj.data);//using data member of A4 class

    }

}

class A4{

    int data=10;

    A4(){

        B b=new B(this);

        b.display();

    }

    public static void main(String args[]){

        A4 a=new A4();

    }

}
```

6. this can be used to return the current class instance from the method.

Syntax:

```
return_type method_name(){

    return this;

}
```

Example:

```
class A{

    A getA(){

        return this;

    }

    void msg(){System.out.println("Hello java");}

}

class Test1{

    public static void main(String args[]){

        new A().getA().msg();

    }

}
```

Example:

```
class Account{

    int a;

    int b;

    public void setData(int a ,int b){

        this.a = a;

        this.b = b;

    }

}
```

```
public void showData(){  
  
    System.out.println("Value of A =" +a);  
  
    System.out.println("Value of B =" +b);  
  
}  
  
public static void main(String args[]){  
  
    Account obj = new Account();  
  
    obj.setData(2,3);  
  
    obj.showData();  
  
}  
  
}
```

Example:

```
class Main {  
  
    int age;  
  
    Main(int age){  
  
        this.age = age;  
  
    }  
  
    public static void main(String[] args) {  
  
        Main obj = new Main(8);  
  
        System.out.println("obj.age = " + obj.age);  
  
    }  
  
}
```

Output:

obj.age= 8

Above program without using this keyword:

```
class Main {  
  
    int age;  
  
    Main(int age){  
  
        age = age;  
  
    }  
  
    public static void main(String[] args) {  
  
        Main obj = new Main(8);  
  
        System.out.println("obj.age = " + obj.age);  
  
    }  
}
```

Output:

obj.age= 0

this keyword with Getters and Setters

```
class Main {  
  
    String name;  
  
    // setter method  
  
    void setName( String name ) {
```

```
        this.name = name;

    }

    // getter method

    String getName(){

        return this.name;

    }

    public static void main( String[] args ) {

        Main obj = new Main();

        // calling the setter and the getter method

        obj.setName("Xyz");

        System.out.println("obj.name: "+obj.getName());

    }

}
```

Output:

```
obj.name: Xyz
```

Method & Constructor Overloading:

Method Overloading: In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java supports polymorphism.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters.

When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Example:

// Demonstrate method overloading.

```
class OverloadDemo {  
  
    void test() {  
  
        System.out.println("No parameters");  
  
        } // Overload test for one integer parameter.  
  
    void test(int a) {  
  
        System.out.println("a: " + a);  
  
        }  
  
    // Overload test for two integer parameters.  
  
    void test(int a, int b) {  
  
        System.out.println("a and b: " + a + " " + b);  
  
        }  
  
}
```

```
    } // overload test for a double parameter

    double test(double a) {

        System.out.println("double a: " + a);

        return a*a;

    }

}

class Overload {

    public static void main(String args[]) {

        OverloadDemo ob = new OverloadDemo();

        double result; // call all versions of test()

        ob.test();

        ob.test(10);

        ob.test(10, 20);

        result = ob.test(123.25);

        System.out.println("Result of ob.test(123.25): " + result);

    }

}
```

Output:

No parameters

a: 10

a and b: 10 20

double a: 123.25

Result of ob.test(123.25): 15190.5625

Method Overloading: method having same name and different parameters.

Advantage of method overloading:

- Method overloading increases the readability of the program.

Different ways to overload the method:

1. By changing the number of arguments.
2. By changing the data type.
1. **By changing number of arguments:**

Example:

```
class Adder{  
  
    static int add(int a,int b){return a+b;}  
  
    static int add(int a,int b,int c){return a+b+c;}  
  
}  
  
class TestOverloading1{  
  
    public static void main(String[] args){  
  
        System.out.println(Adder.add(11,11));  
  
        System.out.println(Adder.add(11,11,11));  
  
    }  
  
}
```

2. By changing the data type.

```
class Adder{

    static int add(int a, int b){return a+b;}

    static double add(double a, double b){return a+b;}

}

class TestOverloading2{

    public static void main(String[] args){

        System.out.println(Adder.add(11,11));

        System.out.println(Adder.add(12.3,12.6));

    }

}
```

Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity.

Ex:

```
class Adder{

    static int add(int a,int b){return a+b;}

    static double add(int a,int b){return a+b;}

}

class TestOverloading3{

    public static void main(String[] args){
```

```
System.out.println(Adder.add(11,11));//ambiguity
```

```
}}
```

Constructor Overloading:

- Constructor overloading in Java is a technique of having more than one constructor with different parameter lists.
- They are arranged in a way that each constructor performs a different task.

Ex:

```
class Student5{  
  
    int id;  
  
    String name;  
  
    int age;  
  
    Student5(int i,String n){  
  
        id = i;  
  
        name = n;  
  
    }  
  
    Student5(int i,String n,int a){  
  
        id = i;  
  
        name = n;  
  
        age=a;  
  
    }  
  
    void display(){System.out.println(id+" "+name+" "+age);}
```

```
public static void main(String args[]){  
  
    Student5 s1 = new Student5(111,"C++");  
  
    Student5 s2 = new Student5(222,"JAVA",25);  
  
    s1.display();  
  
    s2.display();  
  
    }  
  
}
```

Using this keyword in Constructor Overloading.

```
class Complex {  
  
    private int a, b;  
  
    // constructor with 2 parameters  
  
    private Complex( int i, int j ){  
  
        this.a = i;  
  
        this.b = j;  
  
    }  
  
    // constructor with single parameter  
  
    private Complex(int i){  
  
        // invokes the constructor with 2 parameters  
  
        this(i, i);  
  
    }  
  
}
```

```
// constructor with no parameter

private Complex(){

    // invokes the constructor with single parameter

    this(0);

}

public String toString(){

    return this.a + " + " + this.b + "i";

}

public static void main(String[] args ) {

    // creating object of Complex class

    // calls the constructor with 2 parameters

    Complex c1 = new Complex(2, 3);

    // calls the constructor with a single parameter

    Complex c2 = new Complex(3);

    // calls the constructor with no parameters

    Complex c3 = new Complex();

    // print objects

    System.out.println(c1);

    System.out.println(c2);

    System.out.println(c3);

}
```

```
}
```

Output:

2 + 3i

3 + 3i

0 + 0i

Garbage Collection

Garbage Collection is the process of reclaiming the runtime unused memory automatically. In other words, it is a way to destroy the unused objects.

We were using the free() function in C language and delete() in C++. But, in java it is performed automatically. So, java provides better memory management.

- Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.
- In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator.
- Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called garbage collection.
- It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- There is no explicit need to destroy objects as in C++.
- Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

Advantage of Garbage Collection:

- It makes java **memory efficient** because the garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

How can an object be unreferred

1) By nulling a reference:

```
Employee e=new Employee();  
  
e=null;
```

2) By assigning a reference to another:

```
Employee e1=new Employee();  
  
Employee e2=new Employee();  
  
e1=e2;//now the first object referred by e1 is available for garbage collection
```

3) By anonymous object:

```
new Employee();
```

finalize()

- The finalize() method is invoked each time before the object is garbage collected.
- This method can be used to perform cleanup processing.
- This method is defined in Object class as:

```
protected void finalize(){ }
```

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use the finalize method to perform cleanup processing (destroying remaining objects).

gc()

- The gc() method is used to invoke the garbage collector to perform cleanup processing.
- The gc() is found in System and Runtime classes.

```
public static void gc(){ }
```

Note: Garbage collection is performed by a daemon thread called Garbage Collector(GC). This thread calls the finalize() method before the object is garbage collected.

Types of Classes:

There are seven types of classes in Java:

1. **Static Class**
2. **Final Class**
3. **Abstract Class**
4. **Concrete Class**
5. **Singleton Class**
6. **POJO Class**
7. **Inner Class**

Static Class:

- It means that a class that is declared as static within another class is known as a static

class.

- Nested static class does not require reference to the outer class. The purpose of a static class is to provide the outline of its inherited class.
- We can make a class static if and only if it is a nested class.
- We can also say that static classes are known as nested classes.

The properties of the static class are:

1. The class has only static members.
2. It cannot access the member (non-static) of the outer class.
3. We cannot create an object of the static class.

StaticClassExample.java

```
public class StaticClassExample
{
    private static String str = "Hello Xyz";

    //nested class which is a Static class

    public static class StaticDemo
    {
        //non-static method of Static class

        public void show() {
            System.out.println(str);
        }
    }
}
```

```
public static void main(String args[])
{
    StaticClassExample.StaticDemo obj = new StaticClassExample.StaticDemo();

    obj.show();
}
}
```

Output:

```
Hello Xyz
```

Final Class:

- The word final means that cannot be changed.
- The **final** class in Java can be declared using the final keyword.
- Once we declare a class as final, the values remain the same throughout the program.
- The purpose of the final class is to make the class **immutable** like the String class.
- It is only a way to make the class immutable. Remember that the **final class cannot be extended**. It also **prevents the class from being sub-classed**.

Ex:

```
final class A {

void printmsg() {

System.out.print("Base class method is executed.");

}

}

class B extends A {
```

```
void printmsg() {  
  
    System.out.print("Derived class method is executed.");  
  
}  
  
}  
  
//main class  
  
public class FinalClassExample {  
  
    public static void main(String[] arg) {  
  
        B obj = new B();  
  
        obj.printmsg();  
  
    }  
  
}
```

Output:

```
/FinalClassExample.java:11: error: cannot inherit from final A class B extends A
```

Abstract Class:

- An abstract class is a that is declared with the keyword **abstract**. The class may or may not contain abstract methods.
- We cannot create an instance of an abstract class but it can be a subclass. These classes are incomplete, so to complete the abstract class we should extend the abstract classes to a concrete class.
- When we declare a subclass as abstract then it is necessary to provide the implementation of abstract methods. Therefore, the subclass must also be declared abstract. We can achieve data hiding by using the abstract class.
- An example of an abstract class is **AbstractMap** class that is a part of the Collections

framework.

Ex:

```
abstract class MathematicalOperations {  
  
    int a=30, b=40;  
  
    //abstract method  
  
    public abstract void add();  
  
}  
  
public class Operation extends MathematicalOperations {  
  
    //definition of abstract method  
  
    public void add() {  
  
        System.out.println("Sum of a and b is: "a+b);  
  
    }  
  
    public static void main(String args[]) {  
  
        MathematicalOperations obj = new Operation();  
  
        obj.add();  
  
    }  
  
}
```

Output:

```
Sum of a and b is: 70
```

Concrete Class:

- These are the regular Java classes. A derived class that provides the basic implementations for all of the methods that are not already implemented in the base class is known as a **concrete** class.
- In other words, it is regular Java classes in which all the methods of an abstract class are implemented.
- We can create an object of the concrete class directly.
- Remember that concrete class and abstract class are not the same.
- A concrete class may extend its parent class. It is used for specific requirements.

ConcreteClassExample.java

//Concrete Class

```
public class ConcreteClassExample {  
  
    //method of the concreted class  
  
    static int product(int a, int b) {  
  
        return a * b;  
  
    }  
  
    public static void main(String args[]) {  
  
        //method calling  
  
        int p = product(6, 8);  
  
        System.out.println("Product of a and b is: " + p);  
  
    }  
  
}
```

Output:

Product of a and b is: 48

Singleton Class

- A class that has only an object at a time is known as a **singleton class**.
- Still, if we are trying to create an instance a second time, that newly created instance points to the first instance.
- If we made any alteration inside the class through any instance, the modification affects the variable of the single instance, also.
- It is usually used to control access while dealing with the database connection and socket programming.

If we want to create a singleton class, do the following:

- Create a private **constructor**.
- Create a static method (by using the lazy initialization) that returns the object of the singleton class.

SingletonClassExample.java

```
public class Singleton {  
  
    private String objectState;  
  
    private static Singleton instance = null;  
  
    private Singleton() throws Exception {  
  
        this.objectState = "Hello Xyz";  
  
    }  
  
    public static Singleton getInstance() {  
  
        if(instance==null) {  
  
            try {
```

```
        instance=new Singleton();

    }

    catch(Exception e) {

        e.printStackTrace();

    }

}

return instance;

}

public String getObjectState() {

    return objectState;

}

public void setObjectState(String objectState) {

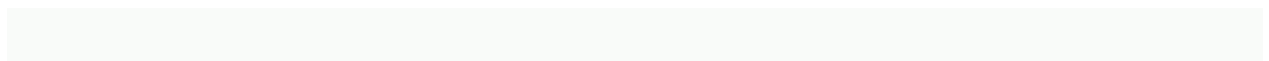
    this.objectState = objectState;

}

}
```

Output:

Hello Xyz



POJO Class:

- In Java, POJO stands for **Plain Old Java Object**.
- A Java class that contains only private variables, setter and getter is known as **POJO** class.
- It is used to define Java objects that increase the reusability and readability of a Java program.
- The class provides encapsulation.

POJO class has the following properties:

- It does not extend the predefined classes such as Arrays, HttpServlet, etc.
- It cannot contain pre-specified annotations.
- It cannot implement predefined interfaces.
- It is not required to add any constructor.
- All instance variables must be private.
- The getter/ setter methods must be public.

PojoClassExample.java

```
class PojoDemo { //private variable

    private double price=89764.34;

    //getter method

    public double getPrice() {

        return price;

    }

    //setter method

    public void setPrice(int price) {

        this.price = price;
```



```
    }  
  
}  
  
//main class  
  
public class PojoClassExample {  
  
    public static void main(String args[]) {  
  
        PojoDemo obj = new PojoDemo();  
  
        System.out.println("The price of an article is "+ obj.getPrice()+" Rs.");  
  
    }  
  
}
```

Output:

The price of an article is 89764.34 Rs.

Inner class:

Java allows us to define a class within a class and such classes are known as **nested classes**. It is used to group the classes logically and to achieve encapsulation. The outer class members (including private) can be accessed by the inner class.

Syntax:

```
class OuterClass {
```

```
//code
```

```
class NestedClass {
```

```
//code
```

```
    }  
}
```

The nested classes are of two types:

1. Static Nested class: A class that is **static** and **nested** is called a static nested class. It interacts with the instance member of its outer class.

Syntax:

```
OuterClass.StaticNestedClass nestedObject = new OuterClass.StaticNestedClass();
```

2. Non-static Nested Class: Non-static nested classes are called **inner classes**.

Syntax:

```
class OuterClass
```

```
{
```

```
    static class StaticNestedClass
```

```
{
```

```
}
```

```
class InnerClass
```

```
{
```

```
...
```

```
}
```

```
}
```

InnerClassExample.java

```
public class InnerClassExample {  
  
    public static void main(String[] args) {  
  
        System.out.println("This is outer class.");  
  
    }  
  
    class InnerClass {  
  
        public void printinner() {  
  
            System.out.println("This is inner class.");  
  
        }  
  
    }  
  
}
```

Types of Inner Classes

Java provides the two types of inner classes are as follows:

- Local Classes or Method Local Inner Class
- Anonymous Classes or Anonymous Inner Class

Local Inner Class:

- It is a type of inner class that is defined inside a block.
- Here block denotes a method body.
- Due to defining inside a block it is also known as method local inner class.
- These are the non-static classes because they can access the instance members of the block.
- We can define the local inner classes in the body of a method. These classes must be instantiated in the block in which they are defined.

- When we compile a program that contains an inner class, the compiler generates the two class files namely **Outer.class** and **Outer\$1Inner.class**. One for the outer class and the other for the inner class that contains a reference to the outer class.

OuterClass.java

```
public class OuterClass {

    private void getValue() {

        int sum = 20;

        //declaring method local inner class

        class InnerClass {

            public int divisor;

            public int remainder;

            public InnerClass() {

                divisor = 4;

                remainder = sum%divisor;

            }

            private int getDivisor() {

                return divisor;

            }

            private int getRemainder() {

                return sum%divisor;

            }

        }

    }

}
```

```
private int getQuotient() {  
  
    System.out.println("We are inside the inner class");  
  
    return sum / divisor;  
  
}  
  
}  
  
//creating an instance of inner class  
  
    InnerClass ic = new InnerClass();  
  
    System.out.println("Divisor = " + ic.getDivisor());  
  
    System.out.println("Remainder = " + ic.getRemainder());  
  
    System.out.println("Quotient = " + ic.getQuotient());  
  
}  
  
public static void main(String[] args)  
  
{  
  
    //creating an instance of outer class  
  
    OuterClass oc = new OuterClass();  
  
    oc.getValue();  
  
}  
  
}
```

Output:

Divisor = 4

Remainder = 0

We are inside the inner class

Quotient = 5

Anonymous Inner Class

It is a type of inner class that is the same as local classes but the only difference is that the class has not a class name and a single object is created of the class. It makes the code more concise. It is used if we want to use the local class once. We can create anonymous classes in the following two ways:

- By using an interface
- By declaring the class concrete and abstract

Syntax:

// the class may an interface, abstract or concrete class

DemoClass obj = **new** DemoClass()

{

//methods

//data members

public void demomethod()

{

//statements

}

};

It is the same as the invocation of constructor except that the class has a definition contained in the block.

AnonymousClassExample.java

```
interface Score

{

    int run = 321;

    void getScore();

}

public class AnonymousClassExample

{

    public static void main(String[] args)

    {

        // Myclass is hidden inner class of Score interface

        // whose name is not written but an object to it

        // is created.

        Score s = new Score()

        {

            @Override

            public void getScore()

            {

                //prints score

                System.out.print("Score is "+run);

            }

        }

    }

}
```

```
};  
  
s.getScore();  
  
}  
  
}
```

Output:

Score is 321

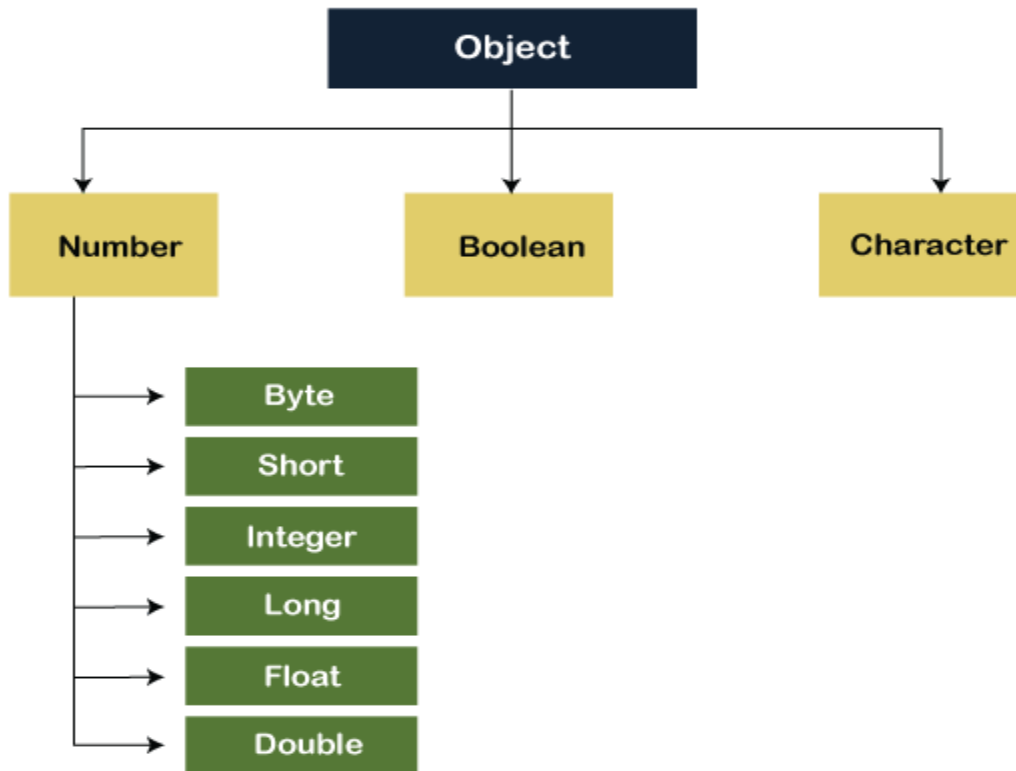
Wrapper Class

In Java, the term wrapper class represents a collection of Java classes that objectify the primitive type of Java.

It means that for each primitive type there is a corresponding wrapper class.

The wrapper classes are used to perform the conversion from a primitive type to an object and vice-versa.

The following figure demonstrates the wrapper class hierarchy.



The following table represents the primitive type and corresponding wrapper class.

Primitive Type	Wrapper Class
Boolean	Boolean
Int	Integer
Char	Character
Double	Double
Float	Float

Long	Long
Byte	Byte
Short	Short

WrapperClassExample.java

```
public class WrapperClassExample
{
    public static void main(String args[])
    {
        byte x = 0;

        //wrapping byte primitive type into Byte object
        Byte byteobj = new Byte(x);

        int y = 23;

        //wrapping int primitive type into Integer object
        Integer intobj = new Integer(y);

        char c='m';

        //wrapping char primitive type into Character object
        Character charobj=c;

        //printing values from objects
```

```
System.out.println("Byte object byteobj: " + byteobj);

System.out.println("Integer object intobj: " + intobj);

System.out.println("Character object charobj: " + charobj);

    }

}
```

Output:

Byte object byteobj: 0

Integer object intobj: 23

Character object charobj: m

Inheritance

Definition: It is a mechanism in which one object acquires all the properties and behaviors of a parent object.

or

The concept of occurring one class features into another class is called Inheritance.

- We can implement Inheritance in Java by using **extends** keyword.
- It uses IS-A relationship (parent-child relationship).

Syntax: class sub_class **extends** super_class{

```
    -----
}
```

Note: Multiple inheritance in Java is not supported using classes.

Advantages:

1. Code reusability.
2. Runtime polymorphism(Method Overriding).

Terms used in Inheritance:

- **Class:** A class is a group of objects which have common properties.
- **Sub_class/Child_Class/derived class/extended class:** Subclass is a class which inherits the other class.
- **Super class/Parent class/Base class:** Superclass is the class from where a subclass inherits the features.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class.

Example Program

```
class Employee{ //Parent class

    float salary=40000;

}

class Programmer extends Employee{ //child class

    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);  
  
}  
  
}
```

Member Access Rules:

- The ‘.’ operator is also known as member operator; it is used to access the member of a package or a class.

Rules:

- If superclass is public then members can be accessed to subclass.
- If superclass is private then members can't be accessed to subclass.
- If superclass is default then members can be accessed to subclass.
- If a superclass is protected then members can be accessed to subclass.
- If superclass is default then members can be accessed to subclass.

Types of Inheritance:

1. Single Inheritance.
2. Multilevel Inheritance.
3. Hierarchical Inheritance.
4. Multiple Inheritance.
5. Hybrid Inheritance.

Single Level Inheritance:

When a class inherits another class, it is known as a **single inheritance**.

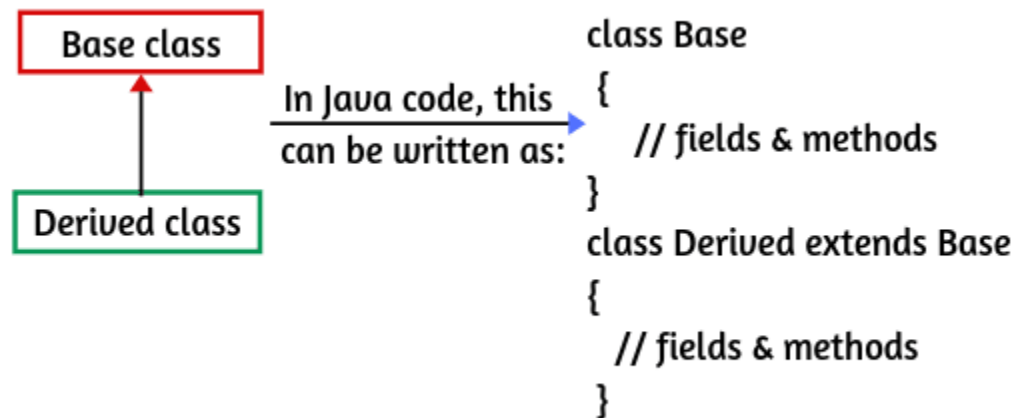


Fig: Single level Inheritance

Ex:

```
class Shape {

    public void display() {

        System.out.println("Inside display");

    }

}

class Rectangle extends Shape {

    public void area() {

        System.out.println("Inside area");

    }

}

public class Tester {

    public static void main(String[] arguments) {
```

```
Rectangle rect = new Rectangle();

rect.display();

rect.area();

}

}
```

Multilevel Inheritance:

In Multilevel Inheritance, a derived class will be inheriting a base class and as well as the derived class also act as the base class to another class.

Syntax: class A{

```
    -----

}

class B extends A{

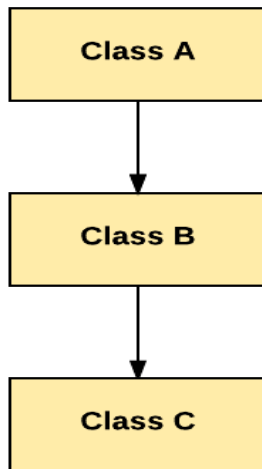
    -----

}

class C extends B{

    -----

}
```



```
Ex: class X {  
  
    public void methodX(){  
  
        System.out.println("Class X method");  
  
    }  
  
}  
  
class Y extends X {  
  
    public void methodY() {  
  
        System.out.println("class Y method");  
  
    }  
  
}
```



```
class Z extends Y {  
  
    public void methodZ(){  
  
        System.out.println("class Z method");  
  
    }  
}  
  
public static void main(String args[]){  
  
    Z obj = new Z();  
  
    obj.methodX();  
  
    obj.methodY(); //calling parent class method  
  
    obj.methodZ(); //calling local method  
  
    }  
}
```

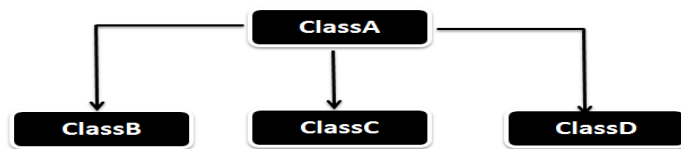
Hierarchical Inheritance.

When two or more classes inherit a single class, it is known as *hierarchical inheritance*.

Syntax:

```
class A{  
  
    -----  
  
}  
  
class B extends A{  
  
    ----- }  
  
class C extends A{
```

```
        ----- }  
  
class D extends A{  
    -----  
  
}
```



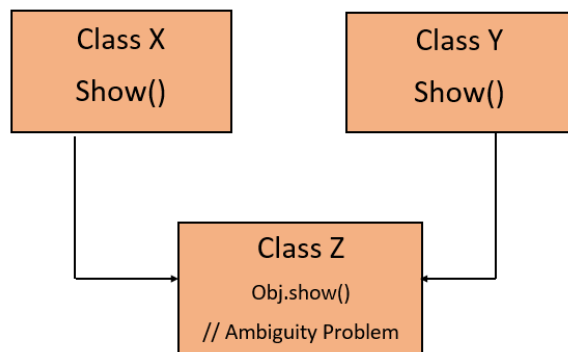
```
Ex: class Animal{  
    void eat(){System.out.println("eating...");}  
  
}  
  
class Dog extends Animal{  
    void bark(){System.out.println("barking...");}  
  
}  
  
class Cat extends Animal{  
    void meow(){System.out.println("meowing...");}  
  
}  
  
class TestInheritance3{  
  
    public static void main(String args[]){  
  
        Cat c=new Cat();  
  
        c.meow();  
    }  
}
```

```
c.eat();  
  
    //c.bark();//C.T.Error  
}  
  
}
```

Multiple Inheritance:

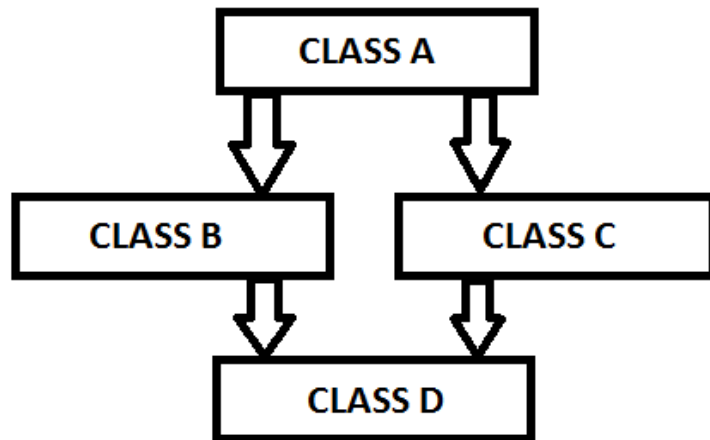
Why is multiple inheritance not supported in java?

- The reason behind this is to prevent ambiguity. Consider a case where class B extends class A and Class C and both class A and C have the same method show(). Now **the Java** compiler cannot decide which display method it should **inherit**. To prevent such a situation, **multiple inheritance is not allowed in java**.



Hybrid Inheritance:

Hybrid Inheritance is a combination of both **Single Inheritance** and **Multiple Inheritance**. Since in **Java Multiple Inheritance** is not supported directly we can achieve Hybrid inheritance also through **Interfaces** only.



Inheritance Examples Programs:

Single Level Inheritance:

```
class Teacher {  
  
    int id;  
  
    String name,  
  
    float sal;  
  
    void setId(int id) {  
  
        this.id=id;  
  
    }  
  
    int getId(){  
  
        return id;  
  
    }  
  
    void setName(String name) {
```

```
        this.name =name;

    }

    String getName() {

        return name;

    }

    void setSal(int sal) {

        this.sal=sal;

    }

    float getSal() {

        return sal;

    }

}

class Student extends Teacher {

    int marks;

    void setmarks(int marks) {

        this.marks= marks;

    }

    int getmarks() {

        return marks;

    }

}

class Inheritance {
```

```
public static void main(String args[]) {  
  
    Student s new =Student(); s.setid(21);  
  
    s.setname("Xyz");  
  
    s.setsal(15000);  
  
    System.out.println("id="+s.getid());  
  
    System.out.println ("id"+s.getname());  
  
    System.out.println ("id="+s.getsal());  
  
    s.setid(21);  
  
    s.setname("Abc");  
  
    s.setmarks(150);  
  
    System.out.println ("id="+s.getid());  
  
    System.out.println ("id"+s.getname());  
  
    System.out.println ("id"+s.getmarks());  
  
    }  
}
```

Single Inheritance with constructor Example:

```
class Single {  
  
    int i,b;  
  
    Single(int l,int b) {  
  
        this.l=l;  
  
        this.b=b;  
  
    }  
}
```

```
}

void formula()

{

    System.out.println("1b" + (1 * b));

}

class Sub extends Single

{

    int h;

    Sub(int l,int b,int hl);

    super(1,b);

    h=hl;

} void volume( ) {

    System.out.println("1bh" + (1 * b * h));

}

}

class SingleInheritance

{

    public static void main(String[] args);

    Sub ss=new Sub(9,7,6);

    ss.formula ();

    ss.volume();

}
```

```
    }  
}
```

Output:

l b=63

l*b*h=378

Single Inheritance:

```
class Room
```

```
{  
  
    void area (int l, int b) {  
  
        System.out.println ("lb-"+(l*b));  
  
    }  
}
```

```
class Bedroom extends Room {
```

```
    void volume(int l, int b, int h) {  
  
        System.out.println("l bh="+ (l*b*h));  
  
    }  
}
```

```
class SingleInheritanceEx {
```

```
    public static void main(String [] args) {  
  
        Bedroom ss= new Bedroom();  
  
        ss.area(5,6);  
  
        ss.volume(4,6,9);  
    }  
}
```



```
    }  
}
```

Output:

1*b=30

1 b*h=216

Multi-Level Inheritance Example Program:

```
class Single {  
    int l,b;  
    Single(int l,int b) {  
        this.l=l;  
        this.b=b;  
    }  
    void formula() {  
        System.out.println("1b"+ (lb));  
    }  
}  
  
class Sub extends Single {  
    int h;  
    Sub(int l,int b,int hl) {  
        super(l,b);  
        h= hl;  
    }  
}
```

```
    }

    void volume(){

        System.out.println("1*b*h"+(1*b*h));

    }

    class Multi extends Sub {

        int r;

        Multi(int l, int b, int h, int r);

        Super(l, b,h);

        this.r=r;

    }

    void display() {

        System.out.println(" (1 b b)/r"+((*b* h) /r));

    }

}

class MultilevelInheritance {

    public static void main(String[] args) {

        Multi ss=new Multi(2,3,4,5);

        ss.formula();

        ss. volume();

        ss.display();

    }

}
```

```
}
```

Output:

```
1 b=6
```

```
1 b h=24
```

```
(1 b*h)/r=4
```

Hierarchical Inheritance:

```
class Hier {
```

```
    public void display() {
```

```
        System.out.println ("Super class");
```

```
    }
```

```
class Chal extends Hier {
```

```
    public void display() {
```

```
        System.out.println("Sub class");
```

```
    }
```

```
class Inher extends Hier {
```

```
    public void display() {
```

```
        System.out.println("Inher Sub class");
```

```
    }
```

```
class Hierarchial {
```

```
    public static void main(String[] args) {  
  
        Chal c=new Chal();  
  
        Inher i new Inher();  
  
        c.display1();  
  
        c.display();  
  
        i.display2();  
  
        i.display();  
  
    }  
  
}
```

Output:

Sub Class

Super class

Inher Sub class

Super Class

Hybrid Inheritance:

```
interface A {  
  
    public void methodA();  
  
}
```

```
interface B extends A {  
  
    public void methodB();  
  
}
```

```
}
```

```
interface C extends A {
```

```
    public void methodC();
```

```
}
```

```
class D implements B, C {
```

```
    public void methodA() {
```

```
        System.out.println("MethodA");
```

```
    }
```

```
    public void methodB() {
```

```
        System.out.println("MethodB");
```

```
    }
```

```
    public void methodc() {
```

```
        System.out.println("MethodC");
```

```
    }
```

```
}
```

```
class HybridInheritance {
```

```
    public static void main(String args[]) {
```

```
        D objl= new D();
```

```
        objl.methodA();
```

```
        obj. methodB();
```

```
        objl.methodC();  
    }  
}
```

Output:

MethodA

MethodB

MethodC

super uses:

- It is a reference variable which is used to refer to an immediate parent class object.

Uses:

1. super can be used to invoke immediate parent class method.
 2. super() can be used to invoke immediate parent class constructor.
- By using **super** keyword we can refer to super class members.

```
    super.variable_name;
```

```
    super.method_name();
```

- By using super keyword we can refer to super class constructors.

```
    super(parameters).
```

- By using super keyword we can restrict the overriding of super class methods.

super is used to refer to the immediate parent class instance variable.

```
class Animal{
```

```
String color="white"; }

class Dog extends Animal{

String color="black";

void printColor(){

System.out.println(color);//prints color of Dog class

System.out.println(super.color);//prints color of Animal class

    }

}

class TestSuper1{

public static void main(String args[]){

Dog d=new Dog();

d.printColor();

    }

}
```

super can be used to invoke parent class method

```
class Animal{

void eat(){System.out.println("eating...");}

}

class Dog extends Animal{

void eat(){System.out.println("eating bread...");}
```

```
void bark(){System.out.println("barking...");}
```

```
void work(){
```

```
    super.eat();
```

```
    bark();
```

```
 }
```

```
class TestSuper2{
```

```
    public static void main(String args[]){
```

```
        Dog d=new Dog();
```

```
        d.work();
```

```
    }
```

```
}
```

Another Example:

// Using super to overcome name hiding.

```
class A {
```

```
    int i;
```

```
}// Create a subclass by extending class A.
```

```
class B extends A {
```

```
    int i; // this i hides the i in A
```

```
    B(int a, int b) {
```

```
        super.i = a; // i in A
```

```
        i = b; // i in B
```



```
}  
  
void show() {  
  
    System.out.println("i in superclass: " + super.i);  
  
    System.out.println("i in subclass: " + i);  
  
}  
  
} class UseSuper {  
  
    public static void main(String args[]) {  
  
        B subOb = new B(1, 2);  
  
        subOb.show();  
  
    }  
  
}
```

Output:

i in superclass: 1

i in subclass: 2

Another Example:

```
class One {  
  
    int i=20;  
  
    void display() {  
  
        System.out.println("Super class Method Variable"+i);  
  
    }  
  
}
```

```
}  
  
class Two extends One {  
  
    int i=30;  
  
    void display() {  
  
        System.out.println("Sub class Method Variable "+i);  
  
        super.display0; // super class method  
  
        System.out.println("Super class Method Variable "+super.i); //super class variable  
  
    }  
  
}  
  
class Superkey {  
  
    public static void main(String args[]) {  
  
        Two t= new Two();  
  
        t.display();  
  
    }  
  
}
```

Output:

Sub class Method Variable 30

Super class Method Variable 20

Super class Method Variable 20

super is used to invoke parent class constructor.

```
class Animal{  
  
    Animal(){System.out.println("animal is created");}  
  
}
```

```
class Dog extends Animal{  
  
    Dog(){  
  
        super();  
  
        System.out.println("dog is created");  
  
    }  
  
}
```

```
class TestSuper3{  
  
    public static void main(String args[]){  
  
        Dog d=new Dog();  
  
    }  
  
}
```

Method Overriding

- If a subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.
- Method overriding is used for runtime polymorphism.

Rules for Method Overriding:

1. The method must have the same name as in the parent class
2. The method must have the same parameter as in the parent class.
3. There must be an IS-A relationship (inheritance)

Ex:

```
class Vehicle{

    void run(){System.out.println("Vehicle is running"); //defining a method

}

}

class Bike2 extends Vehicle{

    //defining the same method as in the parent class

    void run(){System.out.println("Bike is running safely");}

    public static void main(String args[]){

        Bike2 obj = new Bike2();//creating object

        obj.run();//calling method

    }

}
```

Ex:

// Using run-time polymorphism.

```
class Figure {

    double dim1;

    double dim2;

    Figure(double a, double b) {

        dim1 = a;

        dim2 = b;

    }

}
```

```
}

double area() {

    System.out.println("Area for Figure is undefined."); return 0;

}

}

class Rectangle extends Figure {

    Rectangle(double a, double b) {

        super(a, b);

    }

    // override area for rectangle

    double area() {

        System.out.println("Inside Area for Rectangle.");

        return dim1 * dim2;

    }

}

class Triangle extends Figure {

    Triangle(double a, double b) {

        super(a, b);

    } // override area for right triangle

    double area() {

        System.out.println("Inside Area for Triangle.");
```

```
        return dim1 * dim2 / 2;

    }

}

class FindAreas {

    public static void main(String args[]) {

        Figure f = new Figure(10, 10);

        Rectangle r = new Rectangle(9, 5);

        Triangle t = new Triangle(10, 8);

        Figure figref; figref = r;

        System.out.println("Area is " + figref.area());

        figref = t;

        System.out.println("Area is " + figref.area());

        figref = f; System.out.println("Area is " + figref.area());

    }

}
```

Output:

Inside Area for Rectangle.

Area is 45 Inside

Area for Triangle.

Area is 40

Area for Figure is undefined.

Area is 0

Through the dual mechanisms of inheritance and run-time polymorphism, it is possible to define one consistent interface that is used by several different, yet related, types of objects. In this case, if an object is derived from Figure, then its area can be obtained by calling `area()`. The interface to this operation is the same no matter what type of figure is being used.

- Private methods are not visible in the Child classes hence overriding concept is not applicable for private methods. Based on our own requirement we can declare the same Parent class private method in child class also. It is valid but not overriding.

```
class Parent
{
    private void methodOne()
    {}
}
class Child extends Parent
{
    private void methodOne()
    {}
}
```

it is valid but not overriding.

Method Overloading Vs. Method Overriding

Method Overloading	Method Overriding
Method overloading is performed <i>within class</i> .	Method overriding occurs <i>in two classes</i> that have IS-A (inheritance) relationship.
In case of method overloading, <i>parameter must be different</i> .	In case of method overriding, <i>parameter must be same</i> .
Method overloading is the example of <i>compile time polymorphism</i> .	Method overriding is the example of <i>run time polymorphism</i> .
Method overloading is used <i>to increase the readability</i> of the program.	Method overriding is used <i>to provide the specific implementation</i> of the method that is already provided by its super class.
Access specifier can be changed.	Access specifier must not be more restrictive than original method(can be less restrictive).
Static methods can be overloaded which means a class can have more than one static method of same name.	Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
private and final methods can be overloaded	private and final methods cannot be overridden.
Return type of method does not matter in case of method overloading, it can be same or different.	<i>Return type must be same or covariant</i> in method overriding.
overloading is a compile-time concept	Overriding is a run-time concept

Difference Between Method Overriding And Method Hiding



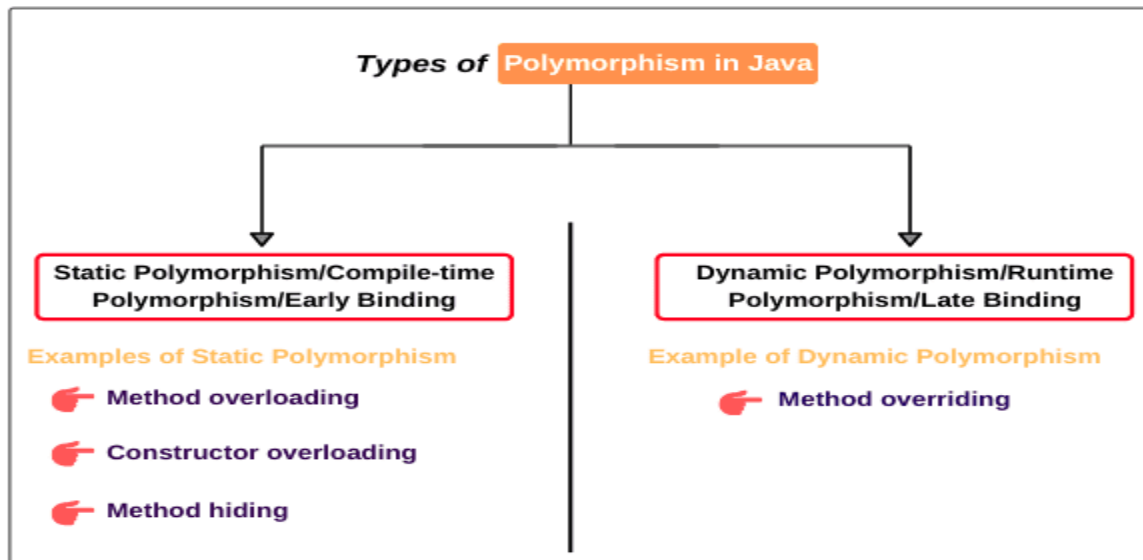
	Method Overriding	Method hiding
1.	Both the methods present in parent and child class are non-static.	Both the methods present in parent and child class are static.
2.	JVM is responsible for resolving method calls based on run-time object.	Compiler is responsible for resolving method calls based on reference type.
3.	It is also called run-time polymorphism/late binding/dynamic polymorphism.	It is also called compile-time polymorphism/early binding/static polymorphism.

CE CodeEaze

Polymorphism

- If one task is performed in **different ways**, it is known as polymorphism.
- Polymorphism is a Greek Word i.e “Poly-**many** Morphism-**different forms**”.

- In Java, we use method overloading and method overriding to achieve polymorphism.



There are two types of binding

1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).

Static binding:

Static binding also called Compile-Time binding or early binding or Method overloading can be achieved in the same class. Compile time polymorphism is nothing but method overloading in java. In simple terms we can say that a class can have more than one methods with same name but with different number of arguments or different types of arguments is known as method overloading.

Example:

```
class Over {  
  
    void area(double l) {
```

```
        System.out.println("area" + (11));

    }

    void area(double l, double b) {

        System.out.println("area" + (l b));

    }

}

class MethodOverloading21 {

    public static void main(String args[]) {

        Over o =new Over();

        o.area(5.3);

        o.area(5.3,4.6);

    }

}
```

Output: area 28.09

The Question is how JVM recognizes which method is called, when both the methods have the same name. For this JVM observes the signature of the methods. Method signature consists of a method name and its parameters. Even if two methods have the same name, their signature may vary. When there is a difference in the method signatures, then the JVM understands both the methods are different and can call the appropriate method.

Dynamic binding:

Dynamic binding also called Run-Time binding or late binding or Method overriding can be achieved in different classes. Dynamic time polymorphism is nothing but the method overriding in java.. In simple terms we can say that a writing two or more methods in super and subclasses such that the methods have same name and same signature is called method overriding

Example:

```
class Riding {  
  
    void Ride(int x) {  
  
        System.out.println("Super class Method"+ (x*x));  
  
    }  
  
}  
  
class Ride extends Riding {  
  
    void Ride(int x) {  
  
        System.out.println("sub class overrides super method"+ (x*x*x));  
  
    }  
  
}  
  
class MethodOverriding {  
  
    public static void main(String[] args) {  
  
        Ride rs=new Ride ();  
  
        rs.Ride(7);  
  
    }  
  
}
```

Output:

sub class overrides super method343

In method overriding, the java compiler does not decide which method is called by the user, since it has to wait till an object to subclass is created. After creating the object, JVM has to bind the method call to an appropriate method. But the methods in super and subclasses have the same name and same method signatures. Then how does the JVM decide which method is called?

Here, JVM calls the methods depending on the class name of the object which is used to call the method. In above program create object to sub class then call the method as shown below

```
rs.show (7);
```

So, the sub class method is only executed by JVM. When a superclass method is overridden by the sub class method, JVM calls only the sub class method and never the super class method. In other words, we can say the sub class method is replacing the super class method.

Static Methods in Polymorphism:

Can we override static methods? No, you cannot override static methods in Java because method overriding is based upon dynamic binding at runtime and static methods are bonded using static binding at compile time. Though you can declare a method with the same name and method signature in sub class which does look like you can override a static method in Java but in reality that is method hiding. Java won't resolve method call at runtime and depending upon the type of Object which is used to call static method, corresponding method will be called. It means if you use Parent class's type to call static method, original static will be called from parent class, on other hand if you use Child class's type to call static method, method from child class will be called. In short you cannot override static method in Java.

If you use Java IDE like Eclipse or Net beans, they will show warning that static method should be called using class name and not by using object because static method cannot be overridden.

Example:

```
class StaticOver {  
  
    static void display( ) {  
  
        System.out.println("Static super class method");  
  
    }  
  
}  
  
class Staticover1 extends StaticOver {  
  
    static void display( ) {  
  
        System.out.println("Static sub class method");  
  
    }  
  
}  
  
class StaticOverriding {  
  
    public static void main(String[] args) {  
  
        StaticOversc= new Staticover1(); // Super class reference refers to sub class object  
        sc.display(); // call display method using super class reference  
  
    }  
  
}
```

Output:

Static super class method

Private Methods in Polymorphism:

Private methods are the methods which are declared by using the access specifier 'private'. This access specifier makes the method not to be available outside the class. So the programmers cannot access the private methods. Even private methods are not available in the sub classes. This means, there is no possibility to override the private methods of the superclass in its subclass. So only method overloading is possible in case of private methods.

Example:

```
class PrivateOver {  
  
    private void display() {  
  
        System.out.println("Private super method");  
  
    }  
  
class PrivateOverl extends PrivateOver {  
  
    private void display() {  
  
        System.out.println("Private sub class method");  
  
    }  
  
class PrivateOverriding {  
  
    public static void main(String[] args) {  
  
        PrivateOverl pv = new PrivateOverl();  
  
        pv.display();  
  
    }  
}
```

```
}
```

Output:

display() has private access in Staticover!

```
sc.display();
```

Dynamic Method Dispatch

- **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
- In this process, an overridden method is called through the reference variable of a superclass.
- The determination of the method to be called is based on the object being referred to by the reference variable.

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

Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.

Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch
```

```
class A {  
  
    void callme() {  
  
        System.out.println("Inside A's callme method");  
  
    }  
  
}  
  
class B extends A {  
  
    // override callme() void callme() {  
  
        System.out.println("Inside B's callme method");  
  
    }  
  
}  
  
class C extends A {  
  
    // override callme()  
  
    void callme() {  
  
        System.out.println("Inside C's callme method");  
  
    }  
  
}  
  
class Dispatch {
```



```
public static void main(String args[]) {  
  
    A a = new A(); // object of type A  
  
    B b = new B(); // object of type B  
  
    C c = new C(); // object of type C  
  
    A r; // obtain a reference of type A  
  
    r = a; // r refers to an A object  
  
    r.callme(); // calls A's version of callme  
  
    r = b; // r refers to a B object  
  
    r.callme(); // calls B's version of callme  
  
    r = c; // r refers to a C object  
  
    r.callme(); // calls C's version of callme  
  
    }  
  
}
```

Output:

Inside A's callme method

Inside B's callme method

Inside C's callme method

This program creates one superclass called A and two subclasses of it, called B and C. Subclasses B and C override callme() declared in A. Inside the main() method, objects of type A, B, and C are declared. Also, a reference of type A, called r, is declared. The program then in turn assigns a reference to each type of object to r and uses that reference to invoke callme(). As the output shows, the version of callme() executed is determined by the type of object being

referred to at the time of the call. Had it been determined by the type of the reference variable, r, you would see three calls to A's callme() method.

- If the reference variable of Parent class refers to the object of Child class, it is known as **upcasting**.

```
class A{ }
```

```
class B extends A{ }
```

```
A a=new B();//upcasting
```

Abstraction:

- Hide internal implementation and just highlight the set of services, is called **abstraction**.
- By using **abstract classes** and **interfaces** we can implement abstraction.
- In real world there are three levels of abstraction:
 - Physical level
 - Conceptual/logical level
 - View level

Ways to achieve Abstraction

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)

Advantages

- We can achieve security as we are not highlighting our internal implementation.
- Enhancement will become very easy because without affecting the end user we are able to perform any type of changes in our internal system.
- It provides more flexibility to the end user to use the system very easily.
- It improves maintainability of the application.

Abstract Classes

A class which is declared with the abstract keyword is known as an abstract class.

- It can have abstract and non-abstract methods (method with the body).
- It can't be instantiated.
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.

Syntax: `abstract class class_name{ }`

Ex: `abstract class Xyz{ }`

Program:

// A Simple demonstration of abstract.

```
abstract class A {  
  
    abstract void callme(); // concrete methods are still allowed in abstract classes  
  
    void callmetoo() {  
  
        System.out.println("This is a concrete method.");  
  
    }  
  
}  
  
class B extends A {  
  
    void callme() {  
  
        System.out.println("B's implementation of callme.");  
  
    }  
  
}
```

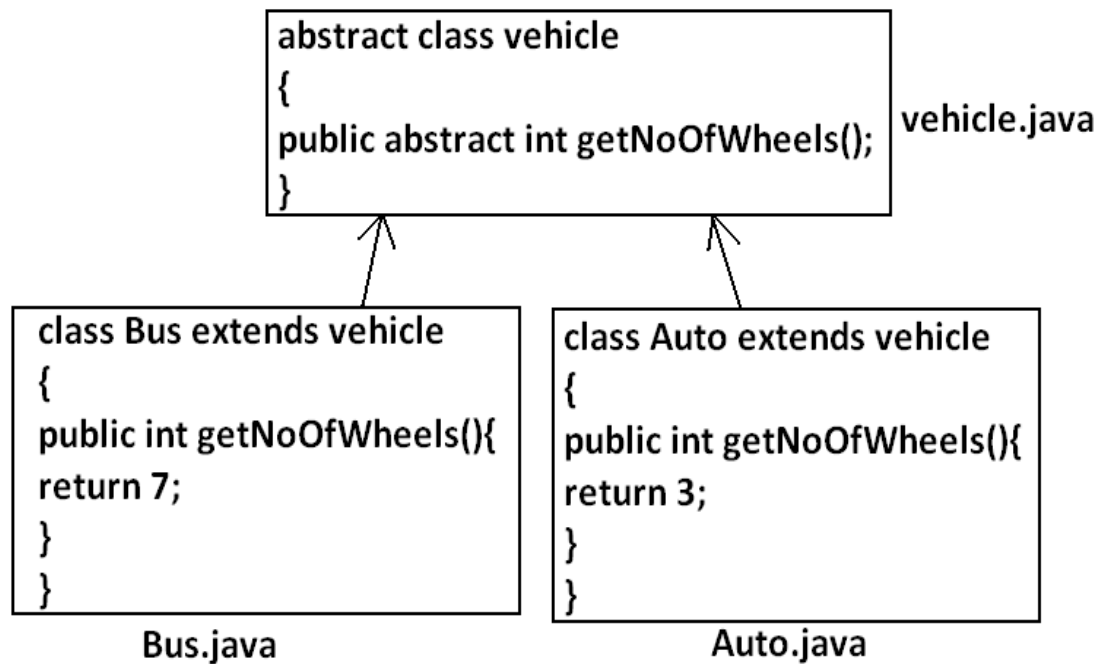
```
class AbstractDemo {  
  
    public static void main(String args[]) {  
  
        B b = new B(); b.callme();  
  
        b.callmetoo();  
  
    }  
  
}
```

Abstract is a modifier applicable only for methods and classes but not for variables.

`public abstract void methodOne();` —————> valid

`public abstract void methodOne(){}` —————> invalid

- Child classes are responsible to provide implementation for parent class abstract methods.



- Not possible to create a method without definition in normal class.

Ex:

```
class Parent
```

```
{
```

```
    public void methodOne();    //C.ERROR
```

```
}
```

- If abstract, No method definition

Ex: class Parent

```
{
```

```
    public abstract void methodOne(){} //C.ERROR
```

```
}
```

- Not possible to create an abstract method in normal class.

Ex:

```
class Parent

{

    public abstract void methodOne();

}
```

- If a class extends any abstract class then it is compulsory we should provide implementation for every abstract method of the parent class otherwise we have to declare child class as abstract.

```
Ex:      abstract class Parent

        {

            public abstract void methodOne();

            public abstract void methodTwo();

        }

        class child extends Parent

        {

            public void methodOne(){ } // C.Error

        }
```

Abstract Method in Java:

A method which is declared as abstract and does not have implementation is known as an

abstract method.

Example:

abstract void printStatus();//no method body and abstract

Example of Abstract class that has an abstract method:

In this example, Bike is an abstract class that contains only one abstract method run. Its implementation is provided by the Honda class.

```
abstract class Bike{

    abstract void run();

}

class Honda4 extends Bike{

void run(){System.out.println("running safely");}

public static void main(String args[]){

    Bike obj = new Honda4();

    obj.run();

    }

}
```

Output:

running safely

Ex Program:

// Using abstract methods and classes.

```
abstract class Figure {  
  
    double dim1;  
  
    double dim2;  
  
    Figure(double a, double b) {  
  
        dim1 = a;  
  
        dim2 = b;  
  
    }  
  
    // area is now an abstract method  
  
    abstract double area();  
  
}  
  
class Rectangle extends Figure {  
  
    Rectangle(double a, double b) {  
  
        super(a, b);  
  
    }  
  
    // override area for rectangle  
  
    double area() {  
  
        System.out.println("Inside Area for Rectangle.");  
  
        return dim1 * dim2;  
  
    }  
  
}
```



```
}

class Triangle extends Figure {

    Triangle(double a, double b) {

        super(a, b);

    }

    // override area for right triangle

    double area() {

        System.out.println("Inside Area for Triangle.");

        return dim1 * dim2 / 2;

    }

}

class AbstractAreas {

    public static void main(String args[]) {

        // Figure f = new Figure(10, 10); // illegal now

        Rectangle r = new Rectangle(9, 5);

        Triangle t = new Triangle(10, 8);

        Figure figref; // this is OK, no object is created

        figref = r;

        System.out.println("Area is " + figref.area());

        figref = t;

        System.out.println("Area is " + figref.area());

    }

}
```

```
    }  
  
}
```

A **factory method** is a method that returns the instance of the class.

Ex:

```
abstract class Shape{  
  
abstract void draw();  
  
}  
  
class Rectangle extends Shape{  
  
void draw(){System.out.println("drawing rectangle");}  
  
}  
  
class Circle1 extends Shape{  
  
void draw(){System.out.println("drawing circle");}  
  
}  
  
class TestAbstraction1 {  
  
public static void main(String args[]){  
  
Shape s=new Circle1();  
  
s.draw();  
  
    }  
  
}
```

Output:

drawing circle

Example Program(abstract class having constructor, data member and methods)

```
abstract class Bike{  
  
    Bike(){System.out.println("bike is created");}  
  
    abstract void run();  
  
    void changeGear(){System.out.println("gear changed");}  
  
}
```

//Creating a Child class which inherits Abstract class

```
class Honda extends Bike{  
  
    void run(){System.out.println("running safely..");}  
  
}  
  
class TestAbstraction2{  
  
    public static void main(String args[]){  
  
        Bike obj = new Honda();  
  
        obj.run();  
  
        obj.changeGear();  
  
    }  
  
}
```

Another example

```
abstract class Bank{
```

```
abstract int getRateOfInterest();

}

class SBI extends Bank{

int getRateOfInterest(){return 7;}

}

class PNB extends Bank{

int getRateOfInterest(){return 8;}

}


class TestBank{

public static void main(String args[]){

Bank b;

b=new SBI();

System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");

b=new PNB();

System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");

}}
```

Output:

Rate of Interest is: 7 %

Note:

- If there is an abstract method in class, that class must be abstract.

- if you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Use of final keyword:

The **final keyword** in java is used to restrict the user. Final can be:

- variable
- method
- class

Final variable

If you make any variable as final, you cannot change the value of final variable

Syntax:

final int x=10;

Example:

```
classFinalex {  
  
public static void main(String[] args) {  
  
final int a=10,b=10;  
  
System.out.println ("Final Variable "+(a+b));  
  
}
```

Output:

Final Variable 20

Another Example:

```
classFinalex {  
  
    public static void main(String[] args) {  
  
        final int a=10,b=10;  
  
        System.out.println ("Final Variable "+ (a+b));  
  
        System.out.println ("Final change variable" + a);  
  
        a=23;  
  
    }  
  
}
```

Output:

Cannot assign a value to final variable a

```
a=11;
```

^1 error

Final method

If you make any method as final, you cannot override it.

Syntax:

```
final void display() {  
  
    //Statements  
  
}
```

Ex:

```
class FinalOver {  
  
    final void display(){  
  
        System.out.println("Final super method");  
  
    }  
  
}  
  
class FinalOverl extends PrivateOver {  
  
    final void display( ) {  
  
        System.out.println("Final sub class method"); }  
  
    }  
  
}  
  
class FinalOverriding {  
  
    public static void main(String[] args) {  
  
        FinalOverl pv new FinalOverl ();  
  
        pv.display();  
  
    }  
  
}
```

Output:

display() in FinalOver cannot override display() in FinalOver, overridden method is final
final void display()

Final class:

If you make any class as final, you cannot extend it.

Syntax:

```
final class A{
```

```
    //stmts
```

```
}
```

```
class B extends A // invalid.
```

Ex:

```
final class FinalOver {
```

```
    final void display() {
```

```
        System.out.println("Final super method");
```

```
    }
```

```
}
```

```
class FinalOverl extends PrivateOver {
```

```
    void display() {
```

```
        System.out.println("Final sub class method"); }
```

```
}
```

```
class FinalOverriding {
```

```
    public static void main(String[] args) {
```

```
        FinalOverl pv = new FinalOverl ();
```

```
        pv.display();
```

```
    }
```

```
}
```


Output:

cannot inherit from final FinalOver

classFinalOver extends FinalOver

^1 error

The final method is inherited but you cannot override it.

For Example:

```
class Bike{  
  
    final void run(){System.out.println("running...");}  
  
}  
  
class Honda2 extends Bike{  
  
    public static void main(String args[]){  
  
        new Honda2().run();  
  
    }  
  
}
```

Output:

running...

final variable	final method	Final class
----------------	--------------	-------------

<pre> class Bike9{ final int speedlimit=90;//final variable void run(){ speedlimit=400; } public static void main(String args[]){ Bike9 obj=new Bike9(); obj.run(); } } //end of clas </pre>	<pre> class Bike{ final void run() {System.out.println("running");} } class Honda extends Bike{ void run(){ System.out.println("running safely with 100kmph");} public static void main(String args[]){ Honda honda= new Honda(); honda.run(); } } </pre>	<pre> final class Bike{ } class Honda1 extends Bike{ void run(){ System.out.println("running safely with 100kmph");} public static void main(String args[]){ Honda1 honda= new Honda1(); honda.run(); } } </pre>
--	--	--

Final Parameter:

Declare any parameter as final, you cannot change the value of it.

```

class Bike11{

int cube(final int n){

n=n+2;//can't be changed as n is final

```

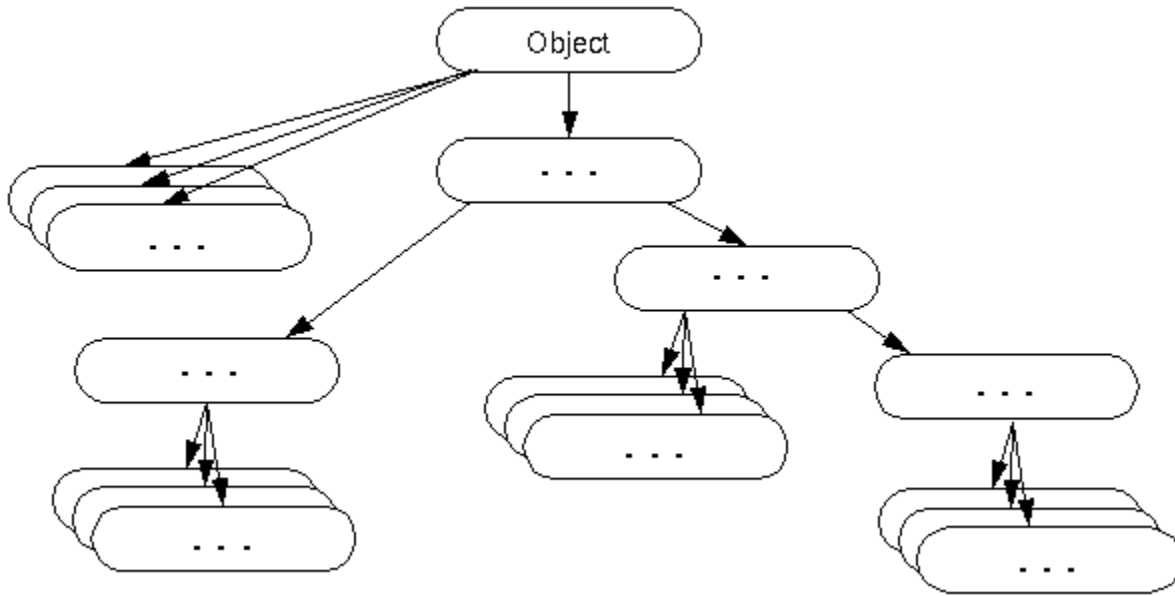
```
n*n*n;  
  
}  
  
public static void main(String args[]){  
  
    Bike11 b=new Bike11();  
  
    b.cube(5);  
  
}  
  
}
```

Output:

Compile time error

Object Class & its Methods

The **Object class** is the parent class of all the classes in java by default.



- **getClass:** This method gives an object that contains the name of class to which an object belongs.

Syntax: **public final** Class getClass()

- **hashCode():** returns the hashcode number of object.

Syntax: **public int** hashCode()

- **equals():** compares the given object to this object.

Syntax: **public boolean** equals(Object obj)

- **clone():** creates and returns the exact copy (clone) of this object.

Syntax: **protected** Object clone() throws CloneNotSupportedException

- **toString:** returns the string representation of this object.

Syntax: **public String** toString()

- **notify():** wakes up single thread, waiting on this object's monitor.

Syntax: **public final void** notify()

- **notifyAll():** wakes up all the threads, waiting on this object's monitor.

Syntax: `public final void notifyAll()`

- **wait():** causes the current thread to wait for the specified **milliseconds**, until another thread notifies (invokes `notify()` or `notifyAll()` method).

Syntax: `public final void wait(long timeout) throws InterruptedException`

- **wait():** causes the current thread to wait for the specified **milliseconds and nanoseconds**, until another thread notifies (invokes `notify()` or `notifyAll()` method).

Syntax: `public final void wait(long timeout, int nanos) throws InterruptedException`

- **wait():** causes the current thread to wait, **until another thread notifies** (invokes `notify()` or `notifyAll()` method).

Syntax: `public final void wait() throws InterruptedException`

- **finalize():** It is invoked by the garbage collector before object is being garbage collected or removed from memory.

Syntax: `protected void finalize() throws Throwable`

Object Cloning:

- **Object cloning** is a way to create an exact copy of an object. The `clone()` method of `Object` class is used to clone an object.
- The **`java.lang.Cloneable` interface** must be implemented by the class whose object clone we want to create.
- If we don't implement a `Cloneable` interface, the `clone()` method generates **`CloneNotSupportedException`**.
- Already an object must present to clone an object.

Syntax:

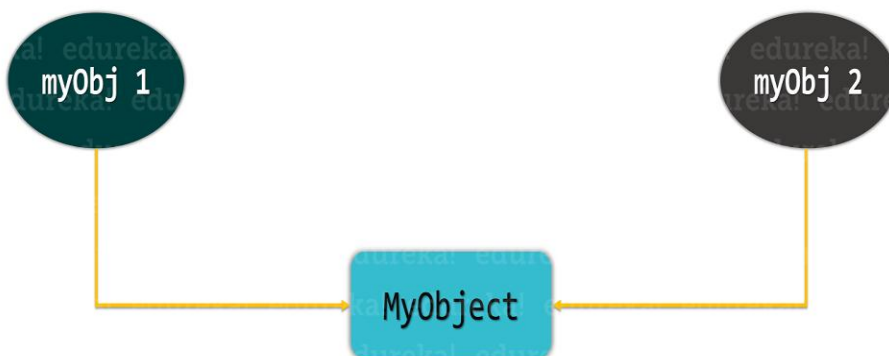
protected Object clone() **throws** CloneNotSupportedException

Types of Cloning:

- **Shallow Clone**
- **Deep clone**

Shallow Clone

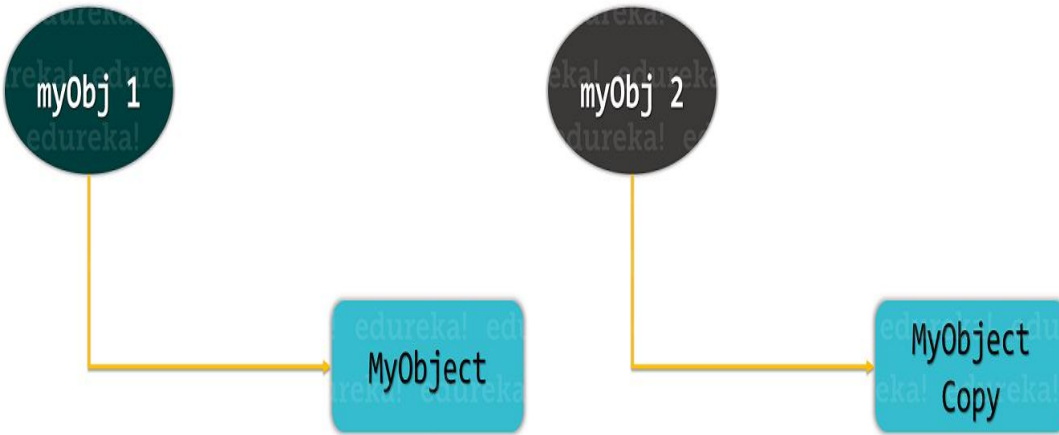
- when the cloning process is done by invoking the clone() method it is called Shallow Cloning.
- It is the default cloning process in Java where a shallow copy of the original object will be created with an exact field.
- if you change the value of the cloned objects then it will be reflected in the original as well.



Deep Clone

- When the cloning process is done by **implementing the Cloneable** interface it is called Deep Cloning.
- In this type of cloning, an exact copy of all the fields of the original object will be

created.



Example:

```
class MyClass{  
    int x;  
    MyClass(int x){  
        this.x=x;  
    }  
}  
  
class Objectex{  
    public static void main(String[] args){  
        MyClass my1= new MyClass(18);  
        MyClass my2 = new MyClass (18);  
    }  
}
```

```
System.out.println(" not equal classobjects 12"+my.hashCode ());

System.out.println(" not equal classobjects 12"+myl.hashCode ());

    Integer il=new Integer (20);

    Integer 12=new Integer (20);

    if(my1.equals (my2))

        System.out.println("equal objects");

    else

        System.out.println("not Equal classobjects");

    if(il.equals(12))

        System.out.println("equal wrapper class objects");

    else

        System.out.println("Not equal wrapper class objects"); }

}
```

Output:

not Equal class objects

equal wrapper class objects

UNIT III

Syllabus:

Interfaces – Interfaces vs. Abstract classes, defining interfaces, implementing and extending interfaces, allowing method definitions in interfaces (Java8).

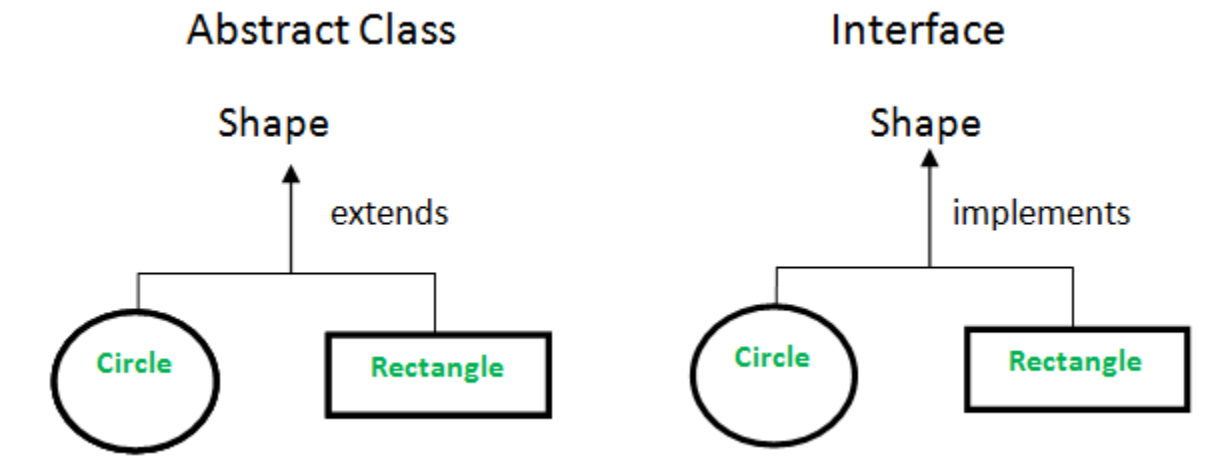
Packages- Defining, Creating and Accessing a Package, Understanding CLASSPATH, importing packages, access protection

Interfaces vs Abstract Classes:

Parameters	Interface	Abstract class
Speed	Slow	Fast
Multiple Inheritances	Implement several Interfaces	Only one abstract class
Structure	Abstract methods	Abstract & concrete methods
When to use	Future enhancement	To avoid independence
Inheritance/ Implementation	A Class can implement multiple interfaces	The class can inherit only one Abstract Class
Access Modifiers	The interface does not have access modifiers. (public by default)	Abstract Class can have an access modifier.
When to use	It is better to use interfaces when various implementations share only method signature. Polymorphic hierarchy of value types.	It should be used when various implementations of the same kind share a common behavior.

Data fields	the interface cannot contain data fields.	the class can have data fields.
Multiple Inheritance Default	A class may implement numerous interfaces.	A class inherits only one abstract class.
Implementation	An interface is abstract so that it can't provide any code.	An abstract class can give complete, default code which should be overridden.
Use of Access modifiers	You cannot use access modifiers for the method, properties, etc.	You can use an abstract class which contains access modifiers.
Usage	Interfaces help to define the peripheral abilities of a class.	An abstract class defines the identity of a class.
Defined fields	No fields can be defined	An abstract class allows you to define both fields and constants
Inheritance	An interface can inherit multiple interfaces but cannot inherit a class.	An abstract class can inherit a class and multiple interfaces.
Constructor or destructors	An interface cannot declare constructors or destructors.	An abstract class can declare constructors and destructors.

Limit of Extensions	It can extend any number of interfaces.	It can extend only one class or one abstract class at a time.
Abstract keyword	In an abstract interface keyword, is optional for declaring a method as an abstract.	In an abstract class, the abstract keyword is compulsory for declaring a method as an abstract.
Class type	An interface can have only public abstract methods.	An abstract class has protected and public abstract methods.
Syntax	<pre>interface <interface_name>{ //methods }</pre>	<pre>abstract class class_name{ // code }</pre>
Example	<pre>interface XYZ{ //method }</pre>	<pre>abstract class XYZ{ //code }</pre>



Defining interface:

- An **interface** is a specification of method prototype. It has static constants and abstract methods.
- The interface in Java is a mechanism to achieve abstraction.
- There can be only abstract methods in the Java interface, not method body.
- It is used to achieve abstraction and multiple inheritance in Java.
- It can't have a method body.

An interface is basically a kind of class. Like classes, interfaces contain methods and variables but with a major difference. The difference is that interfaces declare only abstract methods and final fields. An interface is a specification of method prototypes in java. interface contains only abstract methods and constants. By default all the members in interfaces are public. because interface is implemented in third party vendor classes Note: The methods declared in the interface are defined in the class in which it is implemented.

Syntax:

```
interface <interface_name>{  
  
    // declare constant fields  
  
    // declare methods that abstract  
  
    // by default.  
  
}
```

Example:

```
interface XYZ{  
  
    public void text();  
  
}
```

Variable Declaration in interface: `static final data_type variable_name = value;`

Method Declaration in interface: `return_type method_name(parameter_list);`

Ex:

```
interface XYZ{  
  
    static final int code=0623;  
  
    static final String name="pranay";  
  
    void display();  
  
}
```

- Java Interface also **represents the IS-A relationship**.
- It cannot be instantiated just like the abstract class.
- Since Java 8, we can have **default and static methods** in an interface.

- Since Java 9, we can have **private methods** in an interface.

Uses of interface:

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

Internally Compiler adds the following :

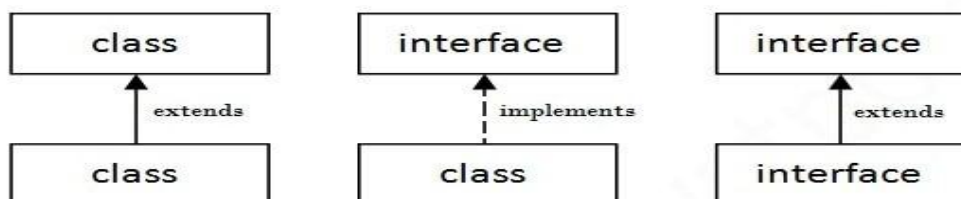
The Java compiler adds public and abstract keywords before the interface method.
Moreover, it adds public, static and final keywords before data members.

Why are the methods of interface public and abstract by default?

- Interface methods are public because they must be available to third party vendors to provide implementation.
- Interface methods are abstract because their implementation is left for third party vendors.

Relationship between Classes & interfaces:

A class extends another class, an interface extends another interface, but a **class implements an interface**.



Implementing in interface:

A class can inherit the members of an interface by implementing it, uses the keyword **implements** **Syntax:**

```
class <classname> implements <Interface_name>
```

```
{
```

```
//body of the class
```

```
}
```

Ex:

```
interface printable{
```

```
    void print();
```

```
}
```

```
class Abc implements printable{
```

```
public void print() {
```

```
    System.out.println("Hello");}
```

```
public static void main(String args[]){
```

```
    Abc obj = new Abc();
```

```
    obj.print();
```

```
}
```

```
}
```

Ex:

```
interface Area {
```

```
    final static float pi = 3.14F;
```

```
    float compute (float x, float y);
```



```
}
```

```
class Rectangle implements Area {  
  
    public float compute (float x, float y) {  
  
        return (x* y);  
  
    }  
  
}
```

```
class Circle implements Area {  
  
    public float compute (float x, float y) {  
  
        return (pi* x* x);  
  
    }  
  
}
```

```
class InterfaceTest {  
  
    public static void main(String args[]) {  
  
        Rectangle rect = new Rectangle ();  
  
        Circle cir=new Circle ();  
  
        Area area;  
  
        area rect;  
  
        System.out.println ("Area of Rectangle = "+area.compute(10, 20));  
  
        area = cir;  
  
        System.out.println ("Area of Circle = "+area.compute (10, 0));  
  
    }  
  
}
```

```
}
```

Output:

Area of rectangle = 200

Area of circle = 314

Extending interface:

An interface can inherit the members of another interface by extending it, using the keyword `extends`.

Syntax:

```
interface name2 extends name1
```

```
{
```

```
    ///body of name2
```

```
}
```

Ex:

```
interface A {
```

```
    void funcA();
```

```
}
```

```
interface B extends A {
```

```
    void funcB();
```

```
}
```

```
class C implements B {
```

```
    public void funcA() {
```

```
        System.out.println("This is funcA");
    }

    public void funcB() {

        System.out.println("This is funcB");

    }

}

public class Demo {

    public static void main(String args[]) {

        C obj = new C();

        obj.funcA();

        obj.funcB();

    }

}
```

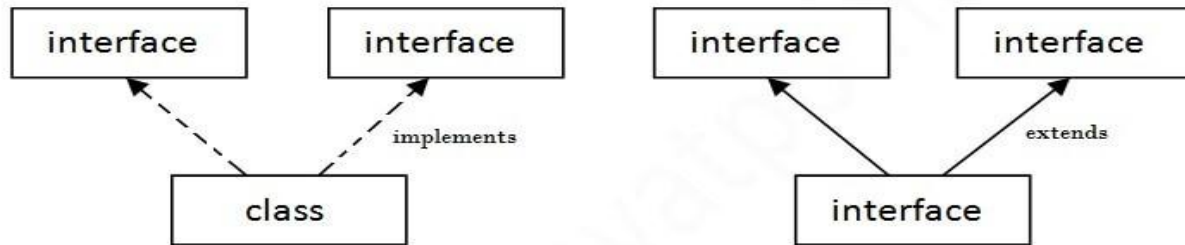
Extending Multiple Interfaces:

- A Java class can only extend one parent class.
- Multiple inheritance is not allowed.
- Interfaces are not classes, however, and an interface can extend more than one parent interface.
- The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

Multiple inheritance in Java by interface

A class implements multiple interfaces or an interface extends multiple interfaces, it is known as

multiple inheritance.



Multiple Inheritance in Java

Ex:

```
interface InterfaceOne {  
    public void disp();  
}  
  
interface InterfaceTwo {  
    public void disp();  
}  
  
public class Main implements InterfaceOne,InterfaceTwo {  
    public void disp() {  
        System.out.println("display() method implementation");  
    }  
  
    public static void main(String args[]) {  
        Main m = new Main();  
    }  
}
```

```
m.disp();  
  
    }  
  
}
```

Accessing Implementations Through Interface References:

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces.

The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the "callee."

This process is similar to using a superclass reference to access a subclass object,

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

```
class Testiface {  
  
    public static void main(String args[]) {  
  
        Callback c = new Client ();  
  
        c.callback (42);  
  
    }  
  
}
```

The output of this program is shown here:callback called with 42 Notice that variable c is declared to be of the interface type Callback, yet it was assigned an instance of Client. Although c can be used to access the callback() method, it cannot access any other members of the Client class. An interface reference variable only has knowledge of the methods declared by its

interface declaration. Thus, c could not be used to access nonfaceMeth() since it is defined by Client but not Callback. While the preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of Callback, Shown here:

```
// Another implementation of Callback. class AnotherClient implements Callback

// Implement Callback's interface

public void callback(int p) {

    System.out.println("Another version of callback");

    System.out.println("p squared is " + (p*p));

}

}

class Testiface2 {

    public static void main(String args[]) {

        Callback c= new Client();

        AnotherClient ob = new AnotherClient();

        c.callback(42);

        c=ob; // c now refers to Another Client object

        c.callback(42);

    }

}
```

Output:

callback called with 42

Another version of callback

p squared is 1764

As you can see, the version of callback() that is called is determined by the type of object that c refers to at run time. While this is a very simple example, you will see another, more practical one shortly.

Allowing Method Definitions in interfaces (Java 8)

Since Java 8, we can have a method body in the interface. But we need to make it the default method. (**Java 8 Default Method in Interface**)

```
interface Drawable{

    void draw();

    default void msg(){System.out.println("default method");}

}

class Rectangle implements Drawable{

    public void draw(){System.out.println("drawing rectangle");}

}

class TestInterfaceDefault{

    public static void main(String args[]){

        Drawable d=new Rectangle();

        d.draw();

        d.msg();

    }

}
```

```
}
```

Java 8 Static Method in Interface:

Since Java 8, we can have static method in interface.

```
interface Drawable{

    void draw();

    static int cube(int x){return x*x*x;}

}

class Rectangle implements Drawable{

    public void draw(){

        System.out.println("drawing rectangle");

    }

}
```

```
class TestInterfaceStatic{

    public static void main(String args[]){

        Drawable d=new Rectangle();

        d.draw();

        System.out.println(Drawable.cube(3));

    }

}
```

Tagged interface:

- An interface which has no member is known as a marker or tagged interface.

- EX: Serializable, Cloneable, Remote, etc.
- They are used to provide some essential information to the JVM, so that JVM may perform some useful operation.

Syntax:

//How is the Serializable interface written?

```
public interface Serializable{  
  
}
```

Nested Interface:

An interface can have another interface which is known as a nested interface.

Ex:

```
interface printable{  
  
    void print();  
  
    interface MessagePrintable{  
  
        void msg();  
  
    }  
  
}
```

Example Program:

```
interface Showable{  
  
    void show();  
  
}
```

```
interface Message{  
  
    void msg();  
  
}
```

```
class TestNestedInterface1 implements Showable.Message  
  
{  
  
    public void msg()  
  
    {  
  
        System.out.println("Hello nested interface");  
  
    }  
  
    public static void main(String args[])  
  
    {  
  
        Showable.Message message=new TestNestedInterface1();//upcasting here  
  
        message.msg();  
  
    }  
  
}
```

class vs interface:

<u>class</u>	<u>interface</u>
class can instantiate variable & create object	interface can't instantiate variable & create

	object.
class can contain concrete methods	the interface can't contain concrete methods.
the access specifiers used with classes are private, protected and public	in interface only public specifier is used

Packages

- A **java package** is a group of similar types of classes, interfaces and sub-packages.
- Packages in java can be categorized in two forms, built-in package and user-defined package.
- There are many built-in packages such as **java, lang, awt, javax, swing, net, io, util, sql** etc.

Packages are Java's way of grouping a variety of classes and/or interfaces together. The grouping is usually done according to functionality. Packages act as "containers" for classes. To create a package is quite easy: simply include a package command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The package statement defines a name space in which classes are stored. If you omit the package statement, the class names are put into the default package, which has no name. (This is why you haven't had to worry about packages before now.) While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

Java uses file system directories to store packages. For example, the .class files for any classes you declare to be part of MyPackage must be stored in a directory called MyPackage. Remember that case is significant, and the directory name must match the package name exactly.

More than one file can include the same package statement. The package statement simply

specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files.

You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as

```
package java.awt.image;
```

needs to be stored in java\awt\image in a Windows environment. Be sure to choose your package names carefully. You cannot rename a package without renaming the directory in which the classes are stored.

Accessing the Package:

The **import** statement is used to search a list of packages for a particular class. The general form of import statement for searching a class is as follows:

Syntax: import packagename.classname;

Steps for accessing the above MyPackage which is created.

Come out of the directory MyPackage

```
cd ..
```

edit the file say DemoPackage.java and the following program

```
import MyPackage.First;
```

```
class DemoPackage {  
  
    public static void main(String args[]) {  
  
        First ob new First ();  
  
        ob.display();  
  
    }  
  
}
```

In a Java source file, import statements occur immediately following the package statement (if it exists) and before any class definitions. This is the general form of the import statement:

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

Here, pkg1 is the name of a top-level package, and pkg2 is the name of a subordinate package inside the outer package separated by a dot (.). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit classname or a star (*), which indicates that the Java compiler should import the entire package. This code fragment shows both forms in use:

```
import java.util.Date;
```

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

All of the standard Java classes included with Java are stored in a package called java. The basic language functions are stored in a package inside of the java package called java.lang. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in java.lang, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:

```
import java.lang.*;
```

If a class with the same name exists in two different packages that you import using the star

form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

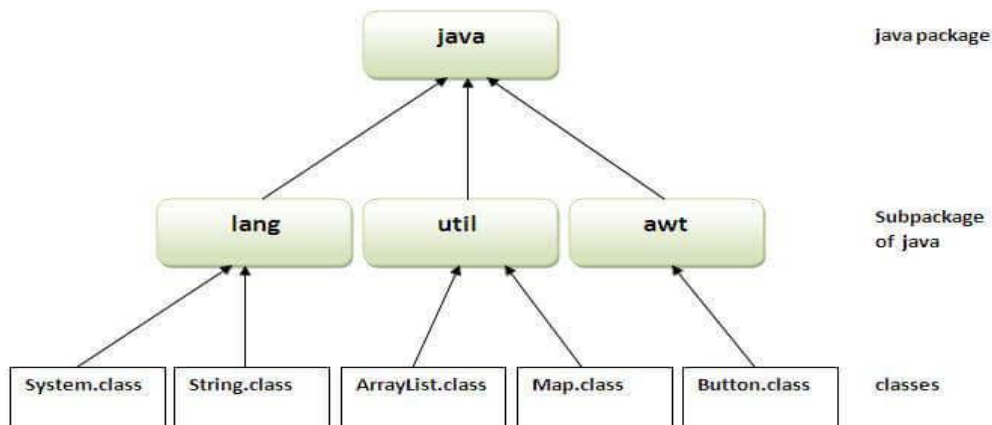
It must be emphasized that the import statement is optional. Any place you use a class name, you can use its fully qualified name, which includes its full package hierarchy. For example, this fragment uses an import statement:

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

```
class MyDate extends Date { }
```

Advantage of Java Package:

- 1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- 2) Java package provides access protection.
- 3) Java package removes naming collisions.



Example:

The **package** keyword is used to create a package in java.

//save as Simple.java

```
package mypack;
```

```
public class Simple{
```

```
    public static void main(String args[]){
```

```
        System.out.println("Welcome to package");
```

```
    }
```

```
}
```

How to compile java package:

syntax:

```
javac -d directory java_filename
```

Example

```
javac -d . Simple.java
```

-d: specifies where to put the specified class file.

How to run java package program

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Creating Package:

Syntax: `package name_of_package;`

Steps:

- Choose the name of the package
- Include the package command as the first line of code in your Java Source File.
- The Source file contains the classes, interfaces, etc you want to include in the package
- Compile to create the Java packages

Step 1: Consider the following package program in Java:

```
package p1;
```

```
class c1(){
```

```
    public void m1(){
```

```
        System.out.println("m1 of c1");
```

```
    }
```

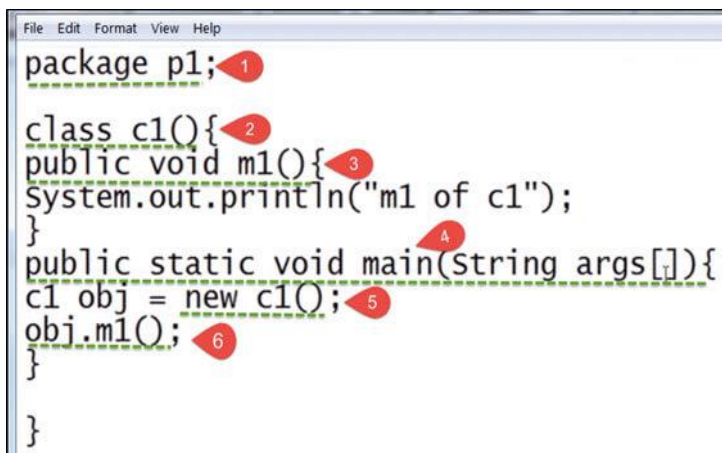
```
public static void main(string args[]){
```

```
    c1 obj = new c1();
```



```
        obj.m1();  
  
    }  
  
}
```

1. To put a class into a package, at the first line of code define package p1.
2. Create a class c1.
3. Defining a method m1 which prints a line.
4. Defining the main method.
5. Creating an object of class c1.
6. Calling method m1.



```
File Edit Format View Help  
package p1;  
  
class c1(){  
    public void m1(){  
        System.out.println("m1 of c1");  
    }  
    public static void main(String args[]){  
        c1 obj = new c1();  
        obj.m1();  
    }  
}
```

Step 2: In next step, save this file as

demo.java

Step 3: In this step, we compile the file.

The compilation is completed. A class file c1 is created. However, no package is created? Next step has the solution

Step 4: Now we have to create a package.

use the command: javac -d . demo.java

This command forces the compiler to create a package. The "." operator represents the current

working directory

Step 5: When you execute the code, it creates a package p1. When you open the java package p1 inside you will see the c1.class file.

Step 6: Compile the same file using the following code

```
javac -d .. demo.java
```

Here ".." indicates the parent directory. In our case the file will be saved in the parent directory which is C Drive. File saved in parent directory when above code is executed.

Step 7: Now let's say you want to create a sub package p2 within our existing java package p1. Then we will modify our code as

```
package p1.p2;  
  
class c1{  
  
public void m1() {  
  
    System.out.println("m1 of c1");  
  
}  
  
}
```

Step 8: Compile the file, it creates a sub-package p2 having class c1 inside the package.

Step 9: To execute the code mention the fully qualified name of the class i.e. the package name followed by the sub-package name followed by the class name -**java p1.p2.c1**. This is how the package is executed and gives the output as "m1 of c1" from the code file.

Accessing a package:

How to access a package from another package?

There are three ways to access the package from outside the package.

1. `import package.*;`
2. `import package.classname;`
3. fully qualified name.

Using `packagename.*`

- If you use `package.*` then all the classes and interfaces of this package will be accessible but not subpackages.
- The `import` keyword is used to make the classes and interface of another package accessible to the current package.

Example:

//save by A.java

```
package pack;
```

```
public class A{
```

```
    public void msg(){System.out.println("Hello");}
```

```
}
```

//save by B.java

```
package mypack;
```

```
import pack.*;
```

```
class B
```

```
{
```

```
    public static void main(String args[])
```

```
{  
  
A obj = new A();  
  
obj.msg();  
  
}  
  
}
```

Using packagename.classname

- If you **import package.class_name** then only declared class of this package will be accessible.

Example

//save by A.java

```
package pack;  
  
public class A{  
  
    public void msg(){System.out.println("Hello");}  
  
}
```

//save by B.java

```
package mypack;  
  
import pack.A;  
  
class B{  
  
    public static void main(String args[]){  
  
        A obj = new A();  
  
        obj.msg();  
  
    }  
}
```

```
}
```

```
}
```

Using fully qualified name:

- If you use a fully qualified name then only declared class of this package will be accessible. Now there is no need to import. But you need to use a fully qualified name every time when you are accessing the class or interface.
- It is generally used when two packages have the same class name e.g. java.util and java.sql packages contain Date class.

Example:

//save by A.java

```
package pack;
```

```
public class A{
```

```
    public void msg() {
```

```
        System.out.println("Hello");}
```

```
}
```

```
package mypack; //save by B.java
```

```
class B{
```

```
    public static void main(String args[]){
```

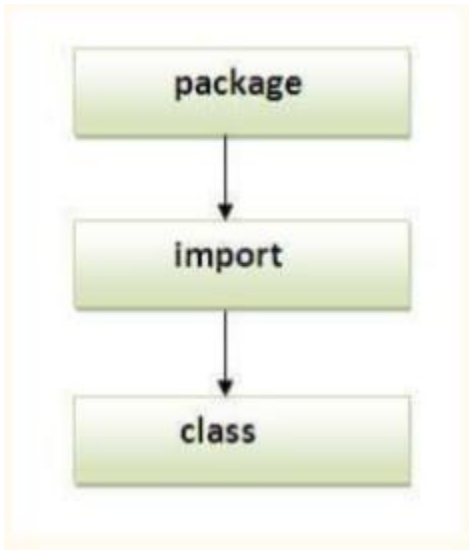
```
        pack.A obj = new pack.A();//using fully qualified name
```

```
        obj.msg(); }
```

```
}
```

Note: If you import a package, subpackages will not be imported.

- Sequence of the program must be package then import then class.



Subpackage in java:

- Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.
- The standard of defining package is **domain.company.package**

Ex. com.javatpoint.bean or org.sssit.dao.

```
package com.javatpoint.core;
```

```
class Simple{
```

```
    public static void main(String args[]){
```

```
        System.out.println("Hello subpackage");
```

```
}
```

```
}
```

Compile: javac -d . Simple.java

Run: java com.javatpoint.core.Simple

CLASSPATH:

- CLASSPATH is an environment variable which is used by Application ClassLoader to locate and load the .class files.
- The CLASSPATH defines the path, to find third-party and user-defined classes that are not extensions or part of Java platform.
- Include all the directories which contain .class files and JAR files when setting the CLASSPATH.

How to Set CLASSPATH in Windows Using Command Prompt

Syntax: set CLASSPATH=%CLASSPATH%;C:\Program Files\Java\jre1.8\rt.jar;

Setting CLASSPATH:

Step 1: Click on the Windows button and choose the Control Panel. Select System.

Step 2: Click on **Advanced System Settings**.

Step 3: A dialog box will open. Click on Environment Variables.

Step 4: If the CLASSPATH already exists in System Variables, click on the Edit button then put a semicolon (;) at the end.

PATH vs CLASSPATH

PATH	CLASSPATH
PATH is an environment variable.	CLASSPATH is also an environment variable.
It is used by the operating system to find the executable files (.exe).	It is used by Application ClassLoader to locate the .class file.
You are required to include the directory which contains .exe files.	You are required to include all the directories which contain .class and JAR files.
PATH environment variable once set, cannot be overridden.	The CLASSPATH environment variable can be overridden by using the command line option -cp or -CLASSPATH to both javac and java command.

ACCESS PROTECTION:

	private	(default)	protected	public
The same class	Yes	Yes	Yes	Yes
Subclass in the package	No	Yes	Yes	Yes
Non-subclass in the package	No	Yes	Yes	Yes
Subclass in another package	No	No	Yes	Yes
Non-subclass in another package	No	No	No	Yes

Java packages can be stored in compressed files called JAR files, allowing classes to download faster as a group rather than one at a time.

In the preceding chapters, you learned about various aspects of Java's access control mechanism and its access specifiers. For example, you already know that access to a **private** member of a class is granted only to other members of that class. Packages add another dimension to

Access control. As you will see, Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other Subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction. Because of the interplay between classes and packages, Java addresses four categories of visibility for class members:

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

The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories. Above table sums up the interactions.

While Java's access control mechanism may seem complicated, we can simplify it as follows.

Anything declared **public** can be accessed from anywhere. Anything declared **private** cannot be seen outside of its class. When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the default access. If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

- A non-nested class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class.

An Access Example:

The following example shows all combinations of the access control modifiers. This example has two packages and five classes. Remember that the classes for the two different packages need to be stored in directories named after their respective packages-in this case, **p1** and **p2**. The source for the first package defines three classes: **Protection**, **Derived** and **SamePackage**. The first class defines four **int** variables in each of the legal protection modes. The variable **n** is declared with the default protection, **n_pri** is private, **n_pro** is protected, and **n_pub** is public.

Each subsequent class in this example will try to access the variables in an instance of this class. The lines that will not compile due to access restrictions are commented out. Before each of these lines is a comment listing the places from which this level of protection would allow access.

The second class, **Derived**, is a subclass of **Protection** in the same package, **p1**. This grants Derived access to every variable in Protection except for **n_pri**, the private one. The third class, **SamePackage**, is not a subclass of **Protection**, but is in the same package and also has access to all but **n_pri**.

Private No Modifier Protected Public

```
package p1;
```

```
public class Protection {  
  
    int n = 1;  
  
    private int n_pri = 2;  
  
    protected int n_pro = 3;  
  
    public int n_pub= 4;  
  
    public Protection() {  
  
        System.out.println("base constructor");  
  
        System.out.println("n + n);  
  
        System.out.println("n_pri"+n_pri);  
  
        System.out.println("n_pro = " +n_pro);  
  
        System.out.println("n_pub= "+n_pub);  
  
    }  
  
}
```

This is file Derived.java:

```
package p1;  
  
class Derived extends Protection {  
  
    Derived() {  
  
        System.out.println("derived constructor");  
  
        System.out.println("n" + n);  
  
        // class only
```

```
//System.out.println("n_pri = "4+n_pri);

System.out.println("n_pro = "+n_pro);

System.out.println("n_pub"+n_pub);

}

}
```

This is file SamePackage.java:

```
package p1;

class SamePackage {

    SamePackage() {

        Protection p= new Protection();

        System.out.println("same package constructor");

        System.out.println("n" + p.n); // class only

        //System.out.println("n_pri" + p.n_pri);

        System.out.println("n_pro-" + p.n_pro);

        System.out.println("n_pub-" +p.n_pub);

    }

}
```

Following is the source code for the other package, **p2**. The two classes defined in p2

cover the other two conditions that are affected by access control. The first class, Protection2, is

a subclass of `pl.Protection`. This grants access to all of `pl.Protection`'s variables except for `n_pri` (because it is private) and `n`, the variable declared with the default protection. Remember, the default only allows access from within the class or the package, not extra-package

subclasses. Finally, the class `OtherPackage` has access to only one variable,

`n_pub`, which was declared public.

This is file `Protection2.java`:

```
package p2;

class Protection2 extends pl.Protection {

    Protection2() {

        System.out.println("derived other package constructor");

        // class or package only

        // System.out.println("n" + n);

        //class only

        // System.out.println("n_pri- " + n_pri);

        System.out.println("n_pro = " + n_pro);

        System.out.println("n_pub-" + n_pub);

    }

}
```

This is file `OtherPackage.java`:

```
package p2;
```

```
class OtherPackage {  
  
    OtherPackage() {  
  
        pl.Protection p = new pl.Protection();  
  
        System.out.println("other package constructor");  
  
        //class or package only  
  
        // System.out.println("n=" + p.n); // class only  
  
        //System.out.println("n_pri"+p.n_pri);  
  
        // class, subclass or package only  
  
        //System.out.println("n_pro = " + p.n_pro);  
  
        System.out.println("n_pub = " + p.n_pub);  
  
    }  
  
}
```

If you wish to try these two packages, here are two test files you can use. The one for package p1 is shown here.

```
// Demo package p1.
```

```
package p1;
```

```
// Instantiate the various classes in p1.
```

```
public class Demo {  
  
    public static void main(String[] args) {  
  
        Protection obl=new Protection();  
  
        Derived ob2=new Derived();  
  
    }  
}
```

```
        SamePackage obd=new SamePackage();

    }

}
```

The test file for p2 is shown next:

```
// Demo package p2.
```

```
package p2;
```

```
// Instantiate the various classes in p2.
```

```
public class Demo {

    public static void main(String[] args) {

        Protection2 obl=new Protection2();

        OtherPackage ob2=new OtherPackage();

    }

}
```

As shown in Table , when a package is imported, only those items within the package declared as public will be available to non-subclasses in the importing code. For example, if you want the Balance class of the package MyPack shown earlier to be available as a stand-alone class for general use outside of MyPack, then you will need to declare it as public and put it into its own file, as shown here:

```
package MyPack;
```

```
/* Now, the Balance class, its constructor, and its show() method are public. This means that
they can be used by non-subclass code outside their package. */
```

```
public class Balance {  
  
    String name;  
  
    double bal;  
  
    public Balance(String n, double b) {  
  
        name = n; bal = b;  
  
    }  
  
    public void show() {  
  
        if(bal<0)  
  
            System.out.print("--> ");  
  
            System.out.println(name + ": $" + bal);  
  
    }  
  
}
```

As you can see, the Balance class is now public. Also, its constructor and its show() method are public, too. This means that they can be accessed by any type of code outside the MyPack package. For example, here TestBalance imports MyPack and is then able to make use of the Balance class:

```
import MyPack.*;  
  
class TestBalance {  
  
    public static void main(String args[]) {  
  
        /* Because Balance is public, you may use Balance class and call its constructor.  
        */  
  
        Balance test = new Balance("Xyz    ", 99.88);  
  
    }  
  
}
```



```
        test.show();

        // you may also call show()

    }

}
```

Types of Package:

1. Built-in packages.
2. User-defined packages.

Built-in packages:

These packages provide almost all necessary classes, interfaces and methods for the programmer to perform any task in the java programs.

1. **java.lang:** lang stands for language. This package provides primary classes and interfaces essential for developing a basic java program.
2. **java.io:** Contains classes for supporting input / output operations. io stands for input and output. This package contains streams. A stream represents the flow of data from one place to another place. Streams are useful to store data in the form of files and also to perform input-output related tasks.
3. **java.util:** util stands for utility. This package contains useful classes and interfaces like Stack, LinkedList, Hashtable, Vector, collections. Arrays, etc. These classes are called collections.
4. **java.applet:** Contains classes for creating Applets. Applets are programs which come from a server into a client and get executed on a client machine on a network.
5. **java.awt:** awt stands for abstract toolkit. This package helps to develop GUI.

programsContain classes for implementing the components for graphical user interfaces (like buttons , ;menus etc).

6. **java.net:** net stands for network. Client-Server Programming can be done by using this package. Contain classes for supporting networking operations.
7. **java.sql:** sql stands for structured query language. This package helps to connect to databases like oracle or Sybase, retrieve the data from them and use it in a java program.
8. **java.text:** This package has two important classes,DateFormat to format dates and times Format which is useful to format numeric values.
9. **javax.swing:** This package helps to develop GUI Programs like java.awt.The x' in javax presents that it is an extended package which means it is a package developed from another package by adding new features to it.

Some important & commonly used packages.

User-defined Packages:

The users of the java language can also create their own packages. They are called user-defined packages. User-defined packages can also be imported into other classes and used exactly in the same way as the built in packages.

Creating a User-defined Package:

We must first declare the name of the package using the package keyword followed by a package name. This must be the first statement in a Java source file.

Syntax:

```
Package DemoPackage;
```

```
// file name: First.java
```

```
package MyPackage;
```

```
    public class First {
```

```
public void display ( ) {  
  
    System.out.println(" Creating the package");  
  
}  
  
}
```

save the program

compile the program: javac-d. First.java

UNIT-IV

Syllabus:

Input/Output exploring of java.io: The Java I/O Classes and Interfaces, File class, The Byte Streams and Character Streams, The Console Class, Using Stream I/O, Serialization

Strings: Strings, string functions,

I/O Basics

Streams

Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Java implements streams within class hierarchies defined in the **java.io** package.

Byte Streams and Character Streams

Java defines two types of streams: byte and character. *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

The original version of Java (Java 1.0) did not include character streams and, thus, all I/O was byte-oriented. Character streams were added by Java 1.1, and certain byte-oriented classes and methods were deprecated. One other point: at the lowest level, all I/O is still byte-oriented. The character-based streams simply provide a convenient and efficient means for handling characters.

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**. Each of these abstract classes has several concrete subclasses that handle the differences between various devices, such as disk files, network connections, and even memory buffers. The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement. Two of the most important are **read()** and **write()**, which, respectively, read and write bytes of data.

The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes, **Reader** and **Writer**. These abstract classes handle Unicode character streams. The abstract classes **Reader** and **Writer** define several key methods that the other stream classes implement. Two of the most important methods are **read()** and **write()**, which read and write characters of data, respectively.

The Predefined Streams

As you know, all Java programs automatically import the **java.lang** package. This package defines a class called **System**, which encapsulates several aspects of the run-time environment. **System** also contains three predefined stream variables: **in**, **out**, and **err**. These fields are declared as **public**, **static**, and **final** within **System**. This means that they can be used by any other part of your program and without reference to a specific **System** object. **System.out** refers to the standard output stream. By default, this is the console. **System.in** refers to standard input, which is the keyboard by default. **System.err** refers to the standard error stream, which also is the console by default. **System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, even though they typically are used to read and write characters from and to the console.

Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements InputStream
FilterOutputStream	Implements OutputStream
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains print() and println()
PushbackInputStream	Input stream that supports one-byte “unget,” which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

The Byte Stream Classes

Stream Class	Meaning
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains print() and println()
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

The Character Stream I/O Classes

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer

The Character Stream I/O Classes

Reading Console Input

In Java 1.0, the only way to perform console input was to use a byte stream, and older code that uses this approach persists. Today, using a byte stream to read console input is still technically possible, but doing so is not recommended. The preferred method of reading console input is to use a character-oriented stream, which makes your program easier to internationalize and maintain.

In Java, console input is accomplished by reading from **System.in**. To obtain a characterbased stream that is attached to the console, wrap **System.in** in a **BufferedReader** object. **BufferedReader** supports a buffered input stream. Its most commonly used constructor is shown here:

BufferedReader(Reader *inputReader*)

Here, *inputReader* is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters. To obtain an **InputStreamReader** object that is linked to **System.in**, use the following constructor:

InputStreamReader(InputStream *inputStream*)

Because **System.in** refers to an object of type **InputStream**, it can be used for *inputStream*. Putting it all together, the following line of code creates a **BufferedReader** that is connected to the keyboard:

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

After this statement executes, **br** is a character-based stream that is linked to the console through **System.in**.

Reading Characters

To read a character from a **BufferedReader**, use **read()**. The version of **read()** that we will be using is :

int read() throws IOException

Each time that **read()** is called, it reads a character from the input stream and returns it as an integer value. It returns **-1** when the end of the stream is encountered. As you can see, it can throw an **IOException**.

// Use a BufferedReader to read characters from the console.

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

```
class BRRead
```

```
{
```

```
public static void main(String args[]) throws IOException
```

```
{
```

```
char c;
```

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

```
System.out.println("Enter characters, 'q' to quit.");
```

```
// read characters
```

```
do {
```

```
c = (char) br.read();
```

```
System.out.println(c);
```

```
} while(c != 'q');
```



```
}  
}
```

Here is a sample run:

Enter characters, 'q' to quit.

123abcq

1

2

3

a

b

c

q

Reading Strings

To read a string from the keyboard, use the version of **readLine()** that is a member of the **BufferedReader** class. Its general form is shown here:

String readLine() throws IOException

As you can see, it returns a **String** object.

// Read a string from console using a BufferedReader.

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

```
class BRReadLines
```

```
{
```

```
public static void main(String args[]) throws IOException
```

```
{
```

```
// create a BufferedReader using System.in
```

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
```

```
String str;
```

```
System.out.println("Enter lines of text.");
```

```
System.out.println("Enter 'stop' to quit.");
```

```
do
```

```
{
```

```
str = br.readLine();
System.out.println(str);
} while(!str.equals("stop"));
}
}
```

PrintWriter Class

For real-world programs, the recommended method of writing to the console when using Java is through a **PrintWriter** stream. **PrintWriter** is one of the character-based classes. Using a character-based class for console output makes it easier to internationalize your program.

PrintWriter defines several constructors. The one we will use is shown here:

```
PrintWriter(OutputStream outputStream, boolean flushOnNewline)
```

Here, *outputStream* is an object of type **OutputStream**, and *flushOnNewline* controls whether Java flushes the output stream every time a **println()** method is called. If *flushOnNewline* is **true**, flushing automatically takes place. If **false**, flushing is not automatic. **PrintWriter** supports the **print()** and **println()** methods for all types including **Object**.

Thus, you can use these methods in the same way as they have been used with **System.out**. If an argument is not a simple type, the **PrintWriter** methods call the object's **toString()** method and then print the result.

Reading Data Using Scanner

The **java.util.Scanner** class was used to read strings and primitive values from the console, “Reading Input from the Console.” A **Scanner** breaks its input into tokens delimited by whitespace characters. To read from the keyboard, you create a **Scanner** for **System.in**, as follows:

```
Scanner input = new Scanner(System.in);
```

To read from a file, create a **Scanner** for a file, as follows:

```
Scanner input = new Scanner(new File(filename));
```

java.util.Scanner	
<pre>+Scanner(source: File) +Scanner(source: String) +close() +hasNext(): boolean +next(): String +nextLine(): String +nextByte(): byte +nextShort(): short +nextInt(): int +nextLong(): long +nextFloat(): float +nextDouble(): double +useDelimiter(pattern: String): Scanner</pre>	<p>Creates a scanner that produces values scanned from the specified file.</p> <p>Creates a scanner that produces values scanned from the specified string.</p> <p>Closes this scanner.</p> <p>Returns true if this scanner has more data to be read.</p> <p>Returns next token as a string from this scanner.</p> <p>Returns a line ending with the line separator from this scanner.</p> <p>Returns next token as a byte from this scanner.</p> <p>Returns next token as a short from this scanner.</p> <p>Returns next token as an int from this scanner.</p> <p>Returns next token as a long from this scanner.</p> <p>Returns next token as a float from this scanner.</p> <p>Returns next token as a double from this scanner.</p> <p>Sets this scanner's delimiting pattern and returns this scanner.</p>

The **Scanner** class contains the methods for scanning data.

Reading and Writing Files

Java provides a number of classes and methods that allow you to read and write files. In Java, all files are byte-oriented, and Java provides methods to read and write bytes from and to a file.

Two of the most often-used stream classes are **FileInputStream** and **FileOutputStream**, which create byte streams linked to files. To open a file, you simply create an object of one of these classes, specifying the name of the file as an argument to the constructor. While both classes support additional, overridden constructors, the following are the forms that we will be using:

`FileInputStream(String fileName)` throws `FileNotFoundException`

`FileOutputStream(String fileName)` throws `FileNotFoundException`

Here, *fileName* specifies the name of the file that you want to open. When you create an input stream, if the file does not exist, then **FileNotFoundException** is thrown. For output streams, if the file cannot be created, then **FileNotFoundException** is thrown. When an output file is opened, any preexisting file by the same name is destroyed.

When you are done with a file, you should close it by calling **close()**. It is defined by both **FileInputStream** and **FileOutputStream**, as shown here:

`void close()` throws `IOException`

To read from a file, you can use a version of **read()** that is defined within **FileInputStream**. The one that we will use is shown here:

`int read()` throws **IOException**

Each time that it is called, it reads a single byte from the file and returns the byte as an integer value. **read()** returns `-1` when the end of the file is encountered. It can throw an **IOException**.

```
import java.util.Scanner;

public class ReadData {
    public static void main(String[] args) throws Exception {
        // Create a File instance
        java.io.File file = new java.io.File("scores.txt");

        // Create a Scanner for the file
        Scanner input = new Scanner(file);

        // Read data from a file
        while (input.hasNext()) {
            String firstName = input.next();
            String mi = input.next();
            String lastName = input.next();
            int score = input.nextInt();
            System.out.println(
                firstName + " " + mi + " " + lastName + " " + score);
        }

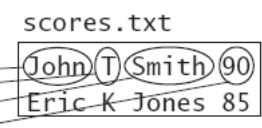
        // Close the file
        input.close();
    }
}
```

create a **File**

create a **Scanner**

has next?
read items

close file



File Content	Scanner Method
John	input.next()
T	input.next()
Smith	input.next()
90	input.nextInt()
Eric	
K	
Jones	
85	

Native Methods

Although it is rare, occasionally you may want to call a subroutine that is written in a language other than Java. Typically, such a subroutine exists as executable code for the CPU and environment in which you are working—that is, native code. For example, you may want to call a native code subroutine to achieve faster execution time. Or, you may want to use a specialized, third-party library, such as a statistical package. However, because Java programs are compiled to bytecode, which is then interpreted (or compiled on-the-fly) by the Java run-time system, it would seem impossible to call a native code subroutine from within your Java program. Fortunately, this conclusion is false. Java provides the **native** keyword,

which is used to declare native code methods. Once declared, these methods can be called from inside your Java program just as you call any other Java method.

To declare a native method, precede the method with the native modifier, but do not define anybody for the method. For example:

```
public native int meth() ;
```

After you declare a native method, you must write the native method and follow a rather complex series of steps to link it with your Java code. Most native methods are written in C. The mechanism used to integrate C code with a Java program is called the *Java Native Interface (JNI)*.

// A simple example that uses a native method.

```
public class NativeDemo
{
    int i;
    public static void main(String args[])
    {
        NativeDemo ob = new NativeDemo();
        ob.i = 10;
        System.out.println("This is ob.i before the native method:" +
            ob.i);
        ob.test(); // call a native method
        System.out.println("This is ob.i after the native method:" +
            ob.i);
    }
    // declare native method
    public native void test() ;
    // load DLL that contains static method
    static
    {
        System.loadLibrary("NativeDemo");
    }
}
```

String Handling

A *string* is a sequence of characters. In many languages, strings are treated as an array of characters, but in Java a string is an object. **String** objects can be constructed a number of ways, making it easy to obtain a string when needed.

When you create a **String** object, you are creating a string that cannot be changed. That is, once a **String** object has been created, you cannot change the characters that comprise that string. At first, this may seem to be a serious restriction. However, such is not the case. You can still perform all types of string operations. The difference is that each time you need an altered version of an existing string, a new **String** object is created that contains the modifications. The original string is left unchanged. This approach is used because fixed, immutable strings can be implemented more efficiently than changeable ones. For those cases in which a modifiable string is desired, Java provides two options: **StringBuffer** and **StringBuilder**. Both hold strings that can be modified after they are created.

The **String**, **StringBuffer**, and **StringBuilder** classes are defined in **java.lang**. Thus, they are available to all programs automatically. All are declared **final**, which means that none of these classes may be subclassed. This allows certain optimizations that increase performance to take place on common string operations. All three implement the **CharSequence** interface. The strings within objects of type **String** are unchangeable means that the contents of the **String** instance cannot be changed after it has been created. However, a variable declared as a **String** reference can be changed to point at some other **String** object at any time.

You can create a string object from a string literal or from an array of characters. To create a string from a string literal, use a syntax like this one:

```
String newString = new String(stringLiteral);
```

The argument `stringLiteral` is a sequence of characters enclosed inside double quotes. The following statement creates a **String** object `message` for the string literal "Welcome to Java":

```
String message = new String("Welcome to Java");
```

Java treats a string literal as a **String** object. So, the following statement is valid:

```
String message = "Welcome to Java";
```

You can also create a string from an array of characters. For example, the following statements create the string "Good Day":

```
char[] charArray = {'G', 'o', 'o', 'd', ' ', 'D', 'a', 'y'};
```

```
String message = new String(charArray);
```

Immutable Strings and Interned Strings

A String object is immutable; its contents cannot be changed. Does the following code change the contents of the string?

```
String s = "Java";  
s = "HTML";
```

The answer is no. The first statement creates a String object with the content “Java” and assigns its reference to s. The second statement creates a new String object with the content “HTML” and assigns its reference to s. The first String object still exists after the assignment, but it can no longer be accessed, because variable s now points to the new object, as shown in Figure:

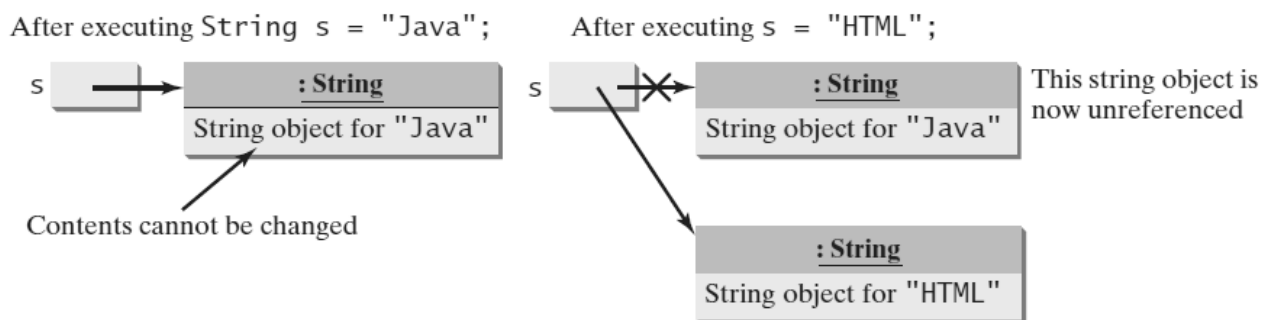
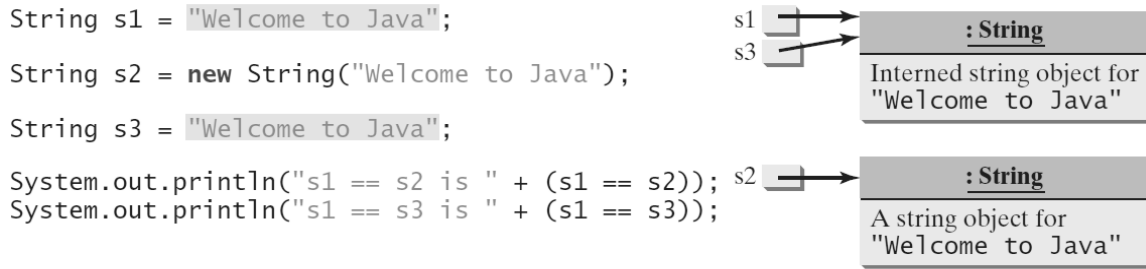


FIGURE : Strings are immutable; once created, their contents cannot be changed.

Since strings are immutable and are ubiquitous in programming, the JVM uses a unique instance for string literals with the same character sequence in order to improve efficiency and save memory. Such an instance is called interned.

In the following example, `s1` and `s3` refer to the same interned string “Welcome to Java”, therefore `s1 == s3` is true. However, `s1 == s2` is false, because `s1` and `s2` are two different string objects, even though they have the same contents.



display

```
s1 == s2 is false
s1 == s3 is true
```

String Comparisons

How do you compare the contents of two strings? You might attempt to use the `==` operator, as follows:

```
if (string1 == string2)
    System.out.println("string1 and string2 are the same object");
else
    System.out.println("string1 and string2 are different objects");
```

However, the `==` operator checks only whether **string1** and **string2** refer to the same object; it does not tell you whether they have the same contents. Therefore, you cannot use the `==` operator to find out whether two string variables have the same contents. Instead, you should use the **equals** method. The code given below, for instance, can be used to compare two strings:

```
if (string1.equals(string2))
    System.out.println("string1 and string2 have the same contents");
else
    System.out.println("string1 and string2 are not equal");
```

For example, the following statements display **true** and then **false**.

```
String s1 = new String("Welcome to Java");
String s2 = "Welcome to Java";
String s3 = "Welcome to C++";
System.out.println(s1.equals(s2)); // true
System.out.println(s1.equals(s3)); // false
```


The **compareTo** method can also be used to compare two strings. For example, consider the following code:

```
s1.compareTo(s2)
```

The method returns the value **0** if **s1** is equal to **s2**, a value less than **0** if **s1** is lexicographically (i.e., in terms of Unicode ordering) less than **s2**, and a value greater than **0** if **s1** is lexicographically greater than **s2**. The actual value returned from the **compareTo** method depends on the offset of the first two distinct characters in **s1** and **s2** from left to right. For example, suppose **s1** is "abc" and **s2** is "abg", and **s1.compareTo(s2)** returns **-4**. The first two characters (**a** vs. **a**) from **s1** and **s2** are compared. Because they are equal, the second two characters (**b** vs. **b**) are compared. Because they are also equal, the third two characters (**c** vs. **g**) are compared. Since the character **c** is **4** less than **g**, the comparison returns **-4**.

Syntax errors will occur if you compare strings by using comparison operators, such as **>**, **>=**, **<**, or **<=**. Instead, you have to use **s1.compareTo(s2)**. The **equals** method returns **true** if two strings are equal and **false** if they are not. The **compareTo** method returns **0**, a positive integer, or a negative integer, depending on whether one string is equal to, greater than, or less than the other string.

java.lang.String	
+equals(s1: String): boolean	Returns true if this string is equal to string s1.
+equalsIgnoreCase(s1: String): boolean	Returns true if this string is equal to string s1 case insensitive.
+compareTo(s1: String): int	Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than s1.
+compareToIgnoreCase(s1: String): int	Same as compareTo except that the comparison is case insensitive.
+regionMatches(index: int, s1: String, s1Index: int, len: int): boolean	Returns true if the specified subregion of this string exactly matches the specified subregion in string s1.
+regionMatches(ignoreCase: boolean, index: int, s1: String, s1Index: int, len: int): boolean	Same as the preceding method except that you can specify whether the match is case sensitive.
+startsWith(prefix: String): boolean	Returns true if this string starts with the specified prefix.
+endsWith(suffix: String): boolean	Returns true if this string ends with the specified suffix.

FIGURE The **String** class contains the methods for comparing strings.

String Length, Characters, and Combining Strings

java.lang.String	
+length(): int	Returns the number of characters in this string.
+charAt(index: int): char	Returns the character at the specified index from this string.
+concat(s1: String): String	Returns a new string that concatenates this string with string s1.

FIGURE The **String** class contains the methods for getting string length, individual characters, and combining strings.

You can get the length of a string by invoking its **length()** method. For example, **message.length()** returns the length of the string **message**. **length** is a method in the **String** class but is a property of an array object. So you have to use **s.length()** to get the number of characters in string **s**, and **a.length** to get the number of elements in array **a**. The **s.charAt(index)** method can be used to retrieve a specific character in a string **s**, where the index is between **0** and **s.length()–1**. For example, **message.charAt(0)** returns the character **W**, as shown in Figure.

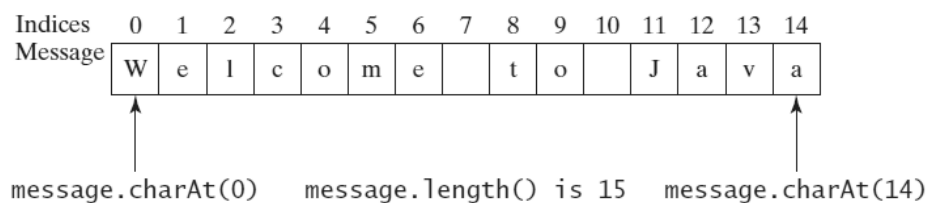


FIGURE A **String** object is represented using an array internally.

When you use a string, you often know its literal value. For convenience, Java allows you to use the string literal to refer directly to strings without creating new variables. Thus, **"Welcome to Java".charAt(0)** is correct and returns **W**. Attempting to access characters in a string **s** out of bounds is a common programming error. To avoid it, make sure that you do not use an index beyond **s.length() – 1**. For example, **s.charAt(s.length())** would cause a **StringIndexOutOfBoundsException**.

Java.io.File Class in Java

The File class is Java's representation of a file or directory path name. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them. The File class contains several methods for working with the path name, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

- It is an abstract representation of file and directory pathnames.
- A pathname, whether abstract or in string form can be either absolute or relative. The parent of an abstract pathname may be obtained by invoking the `getParent()` method of this class.
- First of all, we should create the File class object by passing the filename or directory name to it. A file system may implement restrictions to certain operations on the actual file- system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.
- Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

How to create a File Object?

A File object is created by passing in a String that represents the name of a file, or a String or another File object. For example,

```
File a = new File("/usr/local/bin/geeks");
```

defines an abstract file name for the geeks file in directory /usr/local/bin. This is an absolute abstract file name.

File class constructors:

1) **File f=new File(String name);**

- Creates a java File object that represents name of the file or directory.

2) **File f=new File(String subdirname,String name);**

- Creates a File object that represents name of the file or directory present in specified sub directory.

3) **File f=new File(File subdir,String name);**

Java.io.File Class in Java

The File class is Java's representation of a file or directory path name. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them. The File class contains several methods for working with the path name, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

- It is an abstract representation of file and directory pathnames.
- A pathname, whether abstract or in string form can be either absolute or relative. The parent of an abstract pathname may be obtained by invoking the `getParent()` method of this class.
- First of all, we should create the File class object by passing the filename or directory name to it. A file system may implement restrictions to certain operations on the actual file- system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.

- Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

How to create a File Object?

A File object is created by passing in a String that represents the name of a file, or a String or another File object. For example,

```
File a = new File("/usr/local/bin/geeks");
```

defines an abstract file name for the geeks file in directory /usr/local/bin. This is an absolute abstract file name.

Program to check if a file or directory physically exist or not.

```
// In this program, we accepts a file or directory name from
```

```
// command line arguments. Then the program will check if
```

```
// that file or directory physically exist or not and
```

```
// it displays the property of that file or directory.
```

```
*import java.io.File;
```

```
// Displaying file property class fileProperty
```

```
{
```

```
    public static void main(String[] args) {
```

```
        System.out.println("File   name   :"+f.getName());  System.out.println("Path:   "+f.getPath());
```

```
        System.out.println("Absolute           path:"           +f.getAbsolutePath());
```

```
        System.out.println("Parent:"+f.getParent()); System.out.println("Exists :"+f.exists());
```

```
        if(f.exists())
```

```
        {
```

```
            System.out.println("Is writeable:"+f.canWrite()); System.out.println("Is readable"+f.canRead());
```

```
            System.out.println("Is a directory:"+f.isDirectory()); System.out.println("File Size in bytes
```

```
            "+f.length());
```

```
        }
```

```
    }
```

```
}
```

Requirement: Write code to create a file named with demo.txt in current working directory.

Program:

```
import java.io.*;

class FileDemo

{

    public static void main(String[] args)throws IOException

    {

        File f=new File("demo.txt");

        f.createNewFile();

    }

}
```

Requirement: Write code to create a directory named with bhaskar123 in current working directory and create a file named with abc.txt in that directory.

Program:

```
import java.io.*;

class FileDemo

{

    public static void main(String[] args)throws IOException

    {

        File f1=new File("bhaskar123");

        f1.mkdir();

        File f2=new File("bhaskar123","abc.txt");

        f2.createNewFile();

    }

}
```

```
    }  
}
```

Requirement: Write code to create a file named with demo.txt present in c:\xyz folder.

Program:

```
import java.io.*;  
  
class FileDemo  
{  
  
    public static void main(String[] args)throws IOException  
    {  
  
        File f=new File("c:\\bhaskar","demo.txt");  
  
        f.createNewFile();  
  
    }  
}
```

Import methods of file class:

- 1) **boolean exists();**
 - Returns true if the physical file or directory available.
- 2) **boolean createNewFile();**
 - This method 1st checks whether the physical file is already available or not if it is already available then this method simply returns false. If this file is not already available then it will create a new file and returns true
- 3) **boolean mkdir();**
- 4) **boolean isFile();**
 - Returns true if the File object represents a physical file.
- 5) **boolean isDirectory();**
- 6) **String[] list();**
 - It returns the names of all files and subdirectories present in the specified directory.

7) **long length();**

- Returns the no of characters present in the file.

8) **boolean delete();**

- To delete a file or directory.

UNIT-V

Syllabus:

Exception handling: Fundamentals, exception types, usage of try, catch, throw, throws and finally, built in exceptions, creating your own exceptions subclasses.

Exception:

An exception is an abnormal condition that arises in a code sequence at run time. An exception is a run time error.

To handle these run time errors, we use exception handling. Java exception handling is managed via five keywords : try, catch, throw, throws and finally.

try block : The set of statements which may cause an exception are written within a try block. Program statements that you want to monitor for exceptions are contained within a try block.

Syntax :

```
try
{
    // block of code to monitor for errors
}
```

catch block : The exception raised in the try block is caught into the catch block. Your code can catch the exception raised by the try block and handle it in some rational manner.

Syntax :

```
catch (ExceptionType1 exOb)
{
    // exception handler for ExceptionType1
}
```



```
catch (ExceptionType2 exOb)

{

    // exception handler for ExceptionType2

}
```

throw block : System generated exceptions are automatically thrown by the Java run-time system. To manually throw an exception, use the keyword **throw**.

Syntax :

```
throw new ExceptionType (<Message String>);
```

throws block : Any exception that is thrown out of a method must be specified as such by a **throws** clause.

finally block : Any code that absolutely must be executed before a method returns is put in a **finally** block.

Syntax :

```
finally

{

    // block of code to be executed before try block ends

}
```

- Write a Java program to illustrate the try and catch blocks.

```
class DemoException

{

    public static void main(String args[])
```

```
{  
  
    int a,b;  
  
    try  
  
    {  
  
        b = 0;  
  
        a = 42 / b;  
  
        System.out.println("This will not be printed");  
  
    }  
  
    catch(ArithmeticException e)  
  
    {  
  
        System.out.println("Division by zero");  
  
    }  
  
    System.out.println("After catch statement");  
  
}
```

Output :

Division by zero

After catch statement

- Write a Java program to illustrate multiple catch blocks.

```
class MultiCatch
{
    public static void main(String args[])
    {
        try
        {
            int a = args.length;

            System.out.println("a = "+a);

            Int b = 42 / a;

            Int c[ ] = { 1 };

            C[42] = 99;

        }

        catch(ArithmeticException e)
        {
            System.out.println("Divide by 0 : "+e);
        }

        catch(ArrayIndexOutOfBoundsException e)
        {
            System.out.println("Array index oob : "+e);
        }

        System.out.println("After try/catch blocks");
    }
}
```

```
}
```

```
}
```

- Write a Java program to illustrate Nested try statements.

```
class NestTry
{
    public static void main(String args[])
    {
        try
        {
            int a = args.length;

            int b = 42 / a;

            System.out.println(" a = "+a);

            try
            {
                if(a == 1)

                    a = a/(a - a);

                if(a == 2)
                {
                    int c[ ] = { 1 };

                    c[42] = 99;

                }
            }
        }
    }
}
```

```
    }

    catch(ArrayIndexOutOfBoundsException e)

    {

        System.out.println("Array index out of bounds : "+e);

    }

}

catch(ArithmeticException e)

{

    System.out.println("Divide by 0 : "+e);

}

} }
```

- Write a Java program to illustrate the finally block.

Class FinallyDemo

```
{

    static void procA( )

    {

        try

        {

            System.out.println("inside procA");

            Throw new RuntimeException("demo");

        }

    }

}
```

```
    }

    finally

    {        System.out.println("procA's finally");        }

    }

    static void procB( )

    {

    try

    {

        System.out.println("inside procB");

        Return;

    }

    finally

    {        System.out.println("procB's finally");        }

    }

    static void procC( )

    {

    try

    {

        System.out.println("inside procC");

    }

    finally
```

```
        {      System.out.println("procC's finally");      }

    }

    public static void main(String args[])

    {

    try

    {      procA( );      }

    catch(Exception e)

    {      System.out.println("Exception caught");      }

    procB( );

    procC( );

    }

}
```

Multi Threading

- Single Tasking: Executing only one task at a time is called single tasking. In this single tasking the microprocessor will be sitting idle for most of the time. This means micro processor time is wasted.
- Multi tasking: Executing more than one task at a time is called multi tasking. Multitasking is of two types:
 - Process Based Multitasking: Executing several programs simultaneously is called processbased multi tasking.
 - Thread Based Multitasking: Executing different parts of the same program simultaneously with the help of a thread is called thread based multitasking.

Advantage of multitasking is utilizing the processor time in an optimum way.

Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread. Thread is a smallest unit of code. Thread is also defined as a subprocess. A Thread sometimes called an execution context or a light weight process.

Uses of Threads:

- Threads are used in designing serverside programs to handle multiple clients at a time.
- Threads are used in games and animations.

The Main Thread

When a Java program starts up, one thread begins running immediately. This is usually called the *main thread* of your program, because it is the one that is executed when your program begins. The main thread is important for two reasons:

- It is the thread from which other “child” threads will be spawned.
- Often, it must be the last thread to finish execution because it performs various shutdown actions.

Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object. To do so, you must obtain a reference to it by calling the method **currentThread()**, which is a **public static** member of **Thread**. Its general form is

shown here:

```
static Thread currentThread( )
```

This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

Let's begin by reviewing the following example:

```
class CurrentThreadDemo
{
public static void main(String args[])
{
Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
// change the name of the thread
t.setName("My Thread");
System.out.println("After name change: " + t);
try
{
for(int n = 5; n > 0; n--) {
System.out.println(n);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted");
}
}
}
```

In this program, a reference to the current thread (the main thread, in this case) is obtained by calling **currentThread()**, and this reference is stored in the local variable **t**. Next, the program displays information about the thread. The program then calls **setName()** to change the internal name of the thread. Information about the thread is then redisplayed. Next, a loop counts down from five, pausing one second between each line. The pause is accomplished

by the **sleep()** method. The argument to **sleep()** specifies the delay period in milliseconds. Notice the **try/catch** block around this loop. The **sleep()** method in **Thread** might throw an **InterruptedException**. This would happen if some other thread wanted to interrupt this sleeping one. This example just prints a message if it gets interrupted. In a real program, you would need to handle this differently. Here is the output generated by this program:

Current thread: Thread[main,5,main]

After name change: Thread[My Thread,5,main]

5

4

3

2

1

The Thread Class and the Runnable Interface

Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**. **Thread** encapsulates a thread of execution. Since you can't directly refer to the ethereal state of a running thread, you will deal with it through its proxy, the **Thread** instance that spawned it. To create a new thread, your program will either extend **Thread** or implement the **Runnable** interface. The **Thread** class defines several methods that help manage threads. The ones that will be used in this chapter are shown here:

Method	Meaning
getName()	Obtain the thread name
getPriority()	Obtain a thread priority
Join	Wait for a thread to terminate
Sleep	Suspend a thread for a period of time
Start	Starts thread by calling its run method
Run	Entry point for the thread

Creating a Thread

In the most general sense, you create a thread by instantiating an object of type **Thread**.

Java defines two ways in which this can be accomplished:

- You can implement the **Runnable** interface.
- You can extend the **Thread** class, itself.

The following two sections look at each method, in turn.

Implementing Runnable

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

Runnable abstracts a unit of executable code. You can construct a thread on any object that implements **Runnable**. To implement **Runnable**, a class need only implement a single method called **run()**, which is declared like this:

```
public void run()
```

Inside **run()**, you will define the code that constitutes the new thread. It is important to understand that **run()** can call other methods, use other classes, and declare variables, just like the main thread can. The only difference is that **run()** establishes the entry point for another, concurrent thread of execution within your program. This thread will end when **run()** returns.

After you create a class that implements **Runnable**, you will instantiate an object of type **Thread** from within that class. **Thread** defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName)
```

In this constructor, *threadOb* is an instance of a class that implements the **Runnable** interface. This defines where execution of the thread will begin. The name of the new thread is specified by *threadName*.

After the new thread is created, it will not start running until you call its **start()** method, which is declared within **Thread**. In essence, **start()** executes a call to **run()**. The **start()** method is shown here:

```
void start()
```

Here is an example that creates a new thread and starts it running:

```
.  
class NewThread implements Runnable {  
    Thread t;  
    NewThread() {  
        t = new Thread(this, "Demo Thread");  
        System.out.println("Child thread: " + t);  
        t.start(); // Start the thread  
    }  
    public void run() {  
        try {  
            for(int i = 5; i > 0; i--) {  
                System.out.println("Child Thread: " + i);  
                Thread.sleep(500);  
            }  
        } catch (InterruptedException e) {  
            System.out.println("Child interrupted.");  
        }  
        System.out.println("Exiting child thread.");  
    }  
}  
class ThreadDemo {  
    public static void main(String args[]) {  
        new NewThread(); // create a new thread  
        try {  
            for(int i = 5; i > 0; i--) {  
                System.out.println("Main Thread: " + i);  
                Thread.sleep(1000);  
            }  
        } catch (InterruptedException e) {  
            System.out.println("Main thread interrupted.");  
        }  
    }  
}
```

```
}  
System.out.println("Main thread exiting.");  
}  
}
```

Inside **NewThread**'s constructor, a new **Thread** object is created by the following statement:

```
t = new Thread(this, "Demo Thread");
```

Passing **this** as the first argument indicates that you want the new thread to call the **run()** method on **this** object. Next, **start()** is called, which starts the thread of execution beginning at the **run()** method. This causes the child thread's **for** loop to begin. After calling **start()**, **NewThread**'s constructor returns to **main()**. When the main thread resumes, it enters its **for** loop. Both threads continue running, sharing the CPU, until their loops finish. The output produced by this program is as follows. (Your output may vary based on processor speed and task load.)

Child thread: Thread[Demo Thread,5,main]

Main Thread: 5

Child Thread: 5

Child Thread: 4

Main Thread: 4

Child Thread: 3

Child Thread: 2

Main Thread: 3

Child Thread: 1

Exiting child thread.

Main Thread: 2

Main Thread: 1

Main thread exiting.

Thread Priorities

Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run. In theory, higher-priority threads get more CPU time than lower-priority threads. In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority. (For example, how an operating system implements multitasking can affect the relative availability of CPU time.) A higher-priority thread can also preempt a lower-priority one. For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lowerpriority thread. In theory, threads of equal priority should get equal access to the CPU. But you need to be careful. Remember, Java is designed to work in a wide range of environments. Some of those environments implement multitasking fundamentally differently than others. For safety, threads that share the same priority should yield control once in a while. This ensures that all threads have a chance to run under a nonpreemptive operating system. In practice, even in nonpreemptive environments, most threads still get a chance to run, because most threads inevitably encounter some blocking situation, such as waiting for I/O. When this happens, the blocked thread is suspended and other threads can run. But, if you want smooth multithreaded execution, you are better off not relying on this. Also, some types of tasks are CPU-intensive. Such threads dominate the CPU. For these types of threads, you want to yield control occasionally so that other threads can run. To set a thread's priority, use the **setPriority()** method, which is a member of **Thread**. This is its general form:

```
final void setPriority(int level)
```

Here, *level* specifies the new priority setting for the calling thread. The value of *level* must be within the range **MIN_PRIORITY** and **MAX_PRIORITY**. Currently, these values are 1 and 10, respectively. To return a thread to default priority, specify **NORM_PRIORITY**, which is currently 5. These priorities are defined as **static final** variables within **Thread**.

You can obtain the current priority setting by calling the **getPriority()** method of **Thread**, shown here:

```
final int getPriority()
```

Implementations of Java may have radically different behavior when it comes to scheduling. The Windows XP/98/NT/2000 versions work, more or less, as you would expect. However, other versions may work quite differently. Most of the inconsistencies arise when you have threads that are relying on preemptive behavior, instead of cooperatively giving up CPU time. The safest way to obtain predictable, cross-platform behavior with Java is to use threads that voluntarily give up control of the CPU.

The following example demonstrates two threads at different priorities, which do not run on a preemptive platform in the same way as they run on a nonpreemptive platform.

One thread is set two levels above the normal priority, as defined by **Thread.NORM_PRIORITY**, and the other is set to two levels below it. The threads are started and allowed to run for ten seconds. Each thread executes a loop, counting the number of iterations. After ten seconds, the main thread stops both threads. The number of times that each thread made it through the loop is then displayed.

```
// Demonstrate thread priorities.
```

```
class clicker implements Runnable {
    long click = 0;
    Thread t;
    private volatile boolean running = true;
    public clicker(int p) {
        t = new Thread(this);
        t.setPriority(p);
    }
    public void run() {
        while (running) {
            click++;
        }
    }
    public void stop() {
        running = false;
    }
    public void start() {
```

```
t.start();
}
}

class HiLoPri {
    public static void main(String args[]) {
        Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
        clicker hi = new clicker(Thread.NORM_PRIORITY + 2);
        clicker lo = new clicker(Thread.NORM_PRIORITY - 2);
        lo.start();
        hi.start();
        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }

        lo.stop();
        hi.stop();
        // Wait for child threads to terminate.
        try {
            hi.t.join();
            lo.t.join();
        } catch (InterruptedException e) {
            System.out.println("InterruptedException caught");
        }
        System.out.println("Low-priority thread: " + lo.click);
        System.out.println("High-priority thread: " + hi.click);
    }
}
```

The output of this program, shown as follows when run under Windows, indicates that the threads did context switch, even though neither voluntarily yielded the CPU nor blocked for I/O. The higher-priority thread got the majority of the CPU time.

Low-priority thread: 4408112

High-priority thread: 589626904

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called *synchronization*. As you will see, Java provides unique, language-level support for it. Key to synchronization is the concept of the monitor (also called a *semaphore*). A *monitor* is an object that is used as a mutually exclusive lock, or *mutex*. Only one thread can *own* a monitor at a given time. When a thread acquires a lock, it is said to have *entered* the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor. These other threads are said to be *waiting* for the monitor. A thread that owns a monitor can reenter the same monitor if it so desires.

If you have worked with synchronization when using other languages, such as C or C++, you know that it can be a bit tricky to use. This is because these languages do not, themselves, support synchronization. Instead, to synchronize threads, your programs need to utilize operating system primitives. Fortunately, because Java implements synchronization through language elements, most of the complexity associated with synchronization has been eliminated. You can synchronize your code in either of two ways. Both involve the use of the **synchronized** keyword, and both are examined here.

Using Synchronized Methods

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

To understand the need for synchronization, let's begin with a simple example that does not use it—but should. The following program has three simple classes. The first one, **Callme**, has a single method named **call()**. The **call()** method takes a **String** parameter called **msg**. This method tries to print the **msg** string inside of square brackets. The interesting thing to notice is that after **call()** prints the opening bracket and the **msg** string, it calls **Thread.sleep(1000)**, which pauses the current thread for one second.

The constructor of the next class, **Caller**, takes a reference to an instance of the **Callme** class and a **String**, which are stored in **target** and **msg**, respectively. The constructor also creates a new thread that will call this object's **run()** method. The thread is started immediately. The **run()** method of **Caller** calls the **call()** method on the **target** instance of **Callme**, passing in the **msg** string. Finally, the **Synch** class starts by creating a single instance of **Callme**, and three instances of **Caller**, each with a unique message string. The same instance of **Callme** is passed to each **Caller**.

// This program is not synchronized.

```
class Callme {
    void call(String msg) {
        System.out.print "[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
    public void run() {
        target.call(msg);
    }
}
```

```
}  
}  
class Synch {  
    public static void main(String args[]) {  
        Callme target = new Callme();  
        Caller ob1 = new Caller(target, "Hello");  
        Caller ob2 = new Caller(target, "Synchronized");  
        Caller ob3 = new Caller(target, "World");  
        // wait for threads to end  
        try {  
            ob1.t.join();  
            ob2.t.join();  
            ob3.t.join();  
        } catch (InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
    }  
}
```

Here is the output produced by this program:

```
Hello[Synchronized[World]  
]  
]
```

As you can see, by calling **sleep()**, the **call()** method allows execution to switch to another thread. This results in the mixed-up output of the three message strings. In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time. This is known as a *race condition*, because the three threads are racing each other to complete the method. This example used **sleep()** to make the effects repeatable and obvious. In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur. This can cause a program to run right

one time and wrong the next. To fix the preceding program, you must *serialize* access to **call()**. That is, you must restrict its access to only one thread at a time. To do this, you simply need to precede **call()**'s definition

with the keyword **synchronized**, as shown here

```
class Callme {  
    synchronized void call(String msg) {
```

This prevents other threads from entering **call()** while another thread is using it. After **synchronized** has been added to **call()**, the output of the program is as follows:

[Hello]

[Synchronized]

[World]

Any time that you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should use the **synchronized** keyword to guard the state from race conditions. Remember, once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance. However, nonsynchronized methods on that instance will continue to be callable.

The **synchronized** Statement

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party, and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

```
synchronized(object) {  
    // statements to be synchronized  
}
```

Here, *object* is a reference to the object being synchronized. A synchronized block ensures

that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor.

Here is an alternative version of the preceding example, using a synchronized block within the **run()** method:

// This program uses a synchronized block.

```
class Callme {
    void call(String msg) {
        System.out.print "[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}

class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }
    // synchronize calls to call()
    public void run() {
        synchronized(target) { // synchronized block
            target.call(msg);
        }
    }
}
```

```
}  
}  
class Synch1 {  
    public static void main(String args[]) {  
        Callme target = new Callme();  
        Caller ob1 = new Caller(target, "Hello");  
        Caller ob2 = new Caller(target, "Synchronized");  
        Caller ob3 = new Caller(target, "World");  
        // wait for threads to end  
        try {  
            ob1.t.join();  
            ob2.t.join();  
            ob3.t.join();  
        } catch (InterruptedException e) {  
            System.out.println("Interrupted");  
        }  
    }  
}
```

Interthread Communication

The preceding examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through interprocess communication. As you will see, this is especially easy in Java. As discussed earlier, multithreading replaces event loop programming by dividing your tasks into discrete, logical units. Threads also provide a secondary benefit: they do away with polling. Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time. For example, consider the classic queuing problem, where one thread is producing some data and another is consuming it. To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data. In a polling

system, the consumer would waste many CPU cycles while it waited for the producer to

produce. Once the producer was finished, it would start polling, wasting more CPU cycles waiting for the consumer to finish, and so on. Clearly, this situation is undesirable.

To avoid polling, Java includes an elegant interprocess communication mechanism via the **wait()**, **notify()**, and **notifyAll()** methods. These methods are implemented as **final** methods in **Object**, so all classes have them. All three methods can be called only from within a **synchronized** context. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:

- **wait()** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify()**.
- **notify()** wakes up a thread that called **wait()** on the same object.
- **notifyAll()** wakes up all the threads that called **wait()** on the same object. One of the threads will be granted access.

These methods are declared within **Object**, as shown here:

```
final void wait() throws InterruptedException
```

```
final void notify()
```

```
final void notifyAll()
```

```
// An incorrect implementation of a producer and consumer.
```

```
class Q {
```

```
    int n;
```

```
    synchronized int get() {
```

```
        System.out.println("Got: " + n);
```

```
        return n;
```

```
    }
```

```
    synchronized void put(int n) {
```

```
        this.n = n;
```

```
        System.out.println("Put: " + n);
```

```
    }
```

```
}
```

```
class Producer implements Runnable {
```

```
    Q q;
```

```
    Producer(Q q) {
```

```
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
}
}
}
class Consumer implements Runnable {
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while(true) {
q.get();
}
}
}
class PC {
public static void main(String args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}
```


As you can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use **wait()** and **notify()** to signal in both directions, as shown here:

// A correct implementation of a producer and consumer.

```
class Q {  
    int n;  
    boolean valueSet = false;  
    synchronized int get() {  
        while(!valueSet)  
            try {  
                wait();  
            } catch(InterruptedException e) {  
                System.out.println("InterruptedException caught");  
            }  
        System.out.println("Got: " + n);  
        valueSet = false;  
        notify();  
        return n;  
    }  
    synchronized void put(int n) {  
        while(valueSet)  
            try {  
                wait();  
            } catch(InterruptedException e) {  
                System.out.println("InterruptedException caught");  
            }  
        this.n = n;  
        valueSet = true;  
        System.out.println("Put: " + n);  
    }  
}
```

```
notify();
}
}
class Producer implements Runnable {
    Q q;
    Producer(Q q) {
        this.q = q;
        new Thread(this, "Producer").start();
    }
    public void run() {
        int i = 0;
        while(true) {
            q.put(i++);
        }
    }
}
class Consumer implements Runnable {
    Q q;
    Consumer(Q q) {
        this.q = q;
        new Thread(this, "Consumer").start();
    }
    public void run() {
        while(true) {
            q.get();
        }
    }
}
class PCFixed {
    public static void main(String args[]) {
        Q q = new Q();
```

```
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
}
}
```

Here is some output from this program, which shows the clean synchronous behavior:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```

Deadlock

A special type of error that you need to avoid that relates specifically to multitasking is *deadlock*, which occurs when two threads have a circular dependency on a pair of synchronized objects. For example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread in X tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread waits forever, because to access X, it would have to release its own lock on Y so that the first thread could complete. Deadlock is a difficult error to debug for two reasons:

- In general, it occurs only rarely, when the two threads time-slice in just the right way.
- It may involve more than two threads and two synchronized objects. (That is, deadlock can occur through a more convoluted sequence of events than just described.)

To understand deadlock fully, it is useful to see it in action. The next example creates two classes, **A** and **B**, with methods **foo()** and **bar()**, respectively, which pause briefly before trying to call a method in the other class. The main class, named **Deadlock**, creates an **A** and a **B** instance, and then starts a second thread to set up the deadlock condition. The **foo()** and **bar()** methods use **sleep()** as a way to force the deadlock condition to occur

Example

```
class A {
    synchronized void foo(B b) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered A.foo");
        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("A Interrupted");
        }
        System.out.println(name + " trying to call B.last()");
        b.last();
    }
    synchronized void last() {
        System.out.println("Inside A.last");
    }
}

class B {
    synchronized void bar(A a) {
        String name = Thread.currentThread().getName();
        System.out.println(name + " entered B.bar");
        try {
            Thread.sleep(1000);
        } catch(Exception e) {
            System.out.println("B Interrupted");
        }
        System.out.println(name + " trying to call A.last()");
        a.last();
    }
    synchronized void last() {
        System.out.println("Inside A.last");
    }
}
```

```
}  
}  
class Deadlock implements Runnable {  
    A a = new A();  
    B b = new B();  
    Deadlock() {  
        Thread.currentThread().setName("MainThread");  
        Thread t = new Thread(this, "RacingThread");  
        t.start();  
        a.foo(b); // get lock on a in this thread.  
        System.out.println("Back in main thread");  
    }  
    public void run() {  
        b.bar(a); // get lock on b in other thread.  
        System.out.println("Back in other thread");  
    }  
    public static void main(String args[]) {  
        new Deadlock();  
    }  
}
```

When you run this program, you will see the output shown here:

MainThread entered A.foo

RacingThread entered B.bar

MainThread trying to call B.last()

RacingThread trying to call A.last()

Unit-VI

Collections in java is a framework that provides an architecture to store and manipulate the group of objects. All the operations that you perform on a data such as searching, sorting, insertion, manipulation, deletion etc. can be performed by Java Collections.

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

What is framework in java

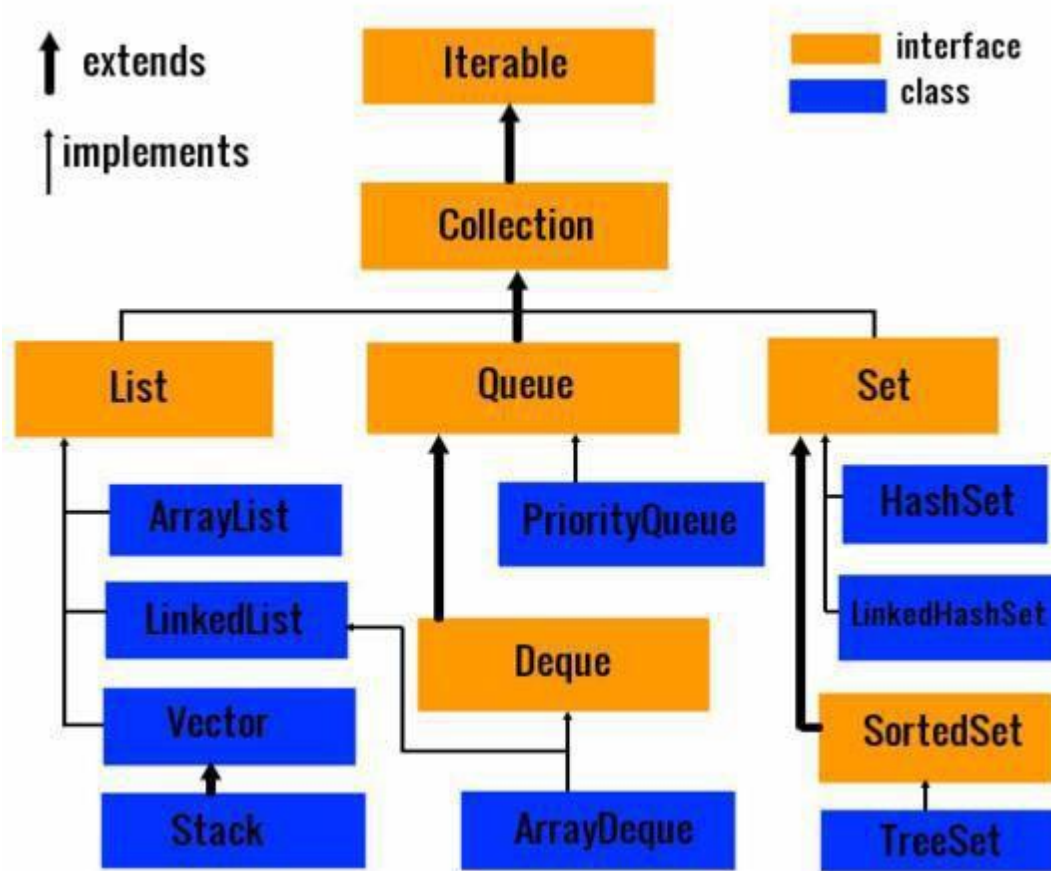
- Provides readymade architecture.
- Represents set of classes and interface.

What is Collection framework

Collection framework represents a unified architecture for storing and manipulating group of objects. It has:

1. Interfaces and its implementations i.e. classes
2. Algorithm

Hierarchy of Collection Framework



Hierarchy of Collection Framework

Java ArrayList class

Java ArrayList class uses a dynamic array for storing the elements. It inherits AbstractList class and implements List interface.

The important points about Java ArrayList class are:

- Java ArrayList class can contain duplicate elements.
- Java ArrayList class maintains insertion order.

- Java ArrayList class is non synchronized.
- Java ArrayList allows random access because array works at the index basis.
- In Java ArrayList class, manipulation is slow because a lot of shifting needs to be occurred if any element is removed from the array list.

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

-
- **Constructors of Java ArrayList**

Constructor	Description
ArrayList()	It is used to build an empty array list.
ArrayList(Collection c)	It is used to build an array list that is initialized with the elements of the collection c.
ArrayList(int capacity)	It is used to build an array list that has the specified initial capacity.

Java ArrayList Example

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

```
class TestCollection1{
```

```
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());

}

}

}
```

vector

ArrayList and Vector both implements List interface and maintains insertion order.

But there are many differences between ArrayList and Vector classes that are given below.

ArrayList	Vector
1) ArrayList is not synchronized.	Vector is synchronized .
2)ArrayList increments 50% of current array size if number of element exceeds from its capacity.	Vector increments 100% means doubles the array size if total number of element exceeds than its capacity.
3)ArrayList is not a legacy class, it is introduced in JDK 1.2.	Vector is a legacy class.
4) ArrayList is fast because it is non-synchronized.	Vector is slow because it is synchronized i.e. in multithreading environment, it will hold the other threads in runnable or non-runnable state until current thread releases the lock of object.
5) ArrayLis tuses Iterator interface to traverse the elements.	Vector uses Enumeration interface to traverse the elements. But it can use Iterator also.

Example of Java Vector

Let's see a simple example of java Vector class that uses Enumeration interface.

```

1. import java.util.*;

2. class TestVector1{

3. public static void main(String args[]){

4. Vector<String> v=new Vector<String>();//creating vector

5. v.add("umesh");//method of Collection

6. v.addElement("irfan");//method of Vector

7. v.addElement("kumar");

```

8. //traversing elements using Enumeration

9. Enumeration e=v.elements();

10. **while**(e.hasMoreElements()){

11. System.out.println(e.nextElement());

12. } } }

Output

umesh

irfan

kumar

Java Hashtable class

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

The important points about Java Hashtable class are:

o A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashCode() method. A Hashtable contains values based on the key.

- It contains only unique elements.
- It may have not have any null key or value.
- It is synchronized.

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.

Constructors of Java Hashtable class

Constructor	Description
Hashtable()	It is the default constructor of hash table it instantiates the Hashtable class.
Hashtable(int size)	It is used to accept an integer parameter and creates a hash table that has an initial size specified by integer value size.
Hashtable(int size, float fillRatio)	It is used to create a hash table that has an initial size specified by size and a fill ratio specified by fillRatio.

Java Hashtable Example

```
import java.util.*;

class TestCollection16{

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
```

Stack

Stack is a subclass of Vector that implements a standard last-in, first-out stack.

Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own.

```
Stack( )
```

Example

```
import java.util.*;  
  
public class StackDemo {  
  
    static void showpush(Stack st, int a) {
```

```
st.push(new Integer(a));

System.out.println("push(" + a + ")");

System.out.println("stack: " + st);}

static void showpop(Stack st) {

System.out.print("pop -> ");

Integer a = (Integer) st.pop();

System.out.println(a);

System.out.println("stack: " + st); }

public static void main(String args[]) {

Stack st = new Stack();

System.out.println("stack: " + st);

showpush(st, 42);

showpush(st, 66);

showpush(st, 99);

showpop(st);

showpop(st);

showpop(st);

try {

showpop(st);

} catch (EmptyStackException e) {

System.out.println("empty    stack");
```

```
}}}
```

This will produce the following
result –

```
stack: [ ] push(42) stack: [42]  
push(66) stack: [42, 66] push(99)
```

```
stack: [42, 66, 99]
```

```
pop -> 99
```

```
stack: [42, 66]
```

```
pop -> 66
```

```
stack: [42]
```

```
pop -> 42 stack: [ ]
```

```
pop -> empty stack
```

Enumeration

The Enumeration interface define the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects. The methods declared by Enumeration are summarized in the following table .

Sr.No.	Method & Description
1	boolean hasMoreElements() When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated.
2	Object nextElement() This returns the next object in the enumeration as a generic Object reference.

Following is shos an example of usage of Enumertaion

This will produce the following result

```
import java.util.Vector;

import java.util.Enumeration;

public class EnumerationTester {

    public static void main(String args[]) {

        Enumeration days;

import java.util.Vector;

import java.util.Enumeration;

public class EnumerationTester {

    public static void main(String args[]) {

        Enumeration days;

        Vector dayNames = new Vector();

        dayNames.add("Sunday");

        ayNames.add("Monday");

        dayNames.add("Tuesday");

        dayNames.add("Wednesday");

        dayNames.add("Thursday");

        dayNames.add("Friday");

        dayNames.add("Saturday");

        days = dayNames.elements();

        while (days.hasMoreElements()) {

            System.out.println(days.nextElement());
```

```
Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday
```

Output

It is a **universal** iterator as we can apply it to any Collection object. By using Iterator, we can perform both read and remove operations. It is improved version of Enumeration with additional functionality of remove-ability of a element.

Iterator must be used whenever we want to enumerate elements in all Collection framework implemented interfaces like Set, List, Queue, Deque and also in all implemented classes of Map interface. Iterator is the **only** cursor available for entire collection framework.

```
// Here "c" is any Collection object. itr is of
// type Iterator interface and refers to "c"
Iterator itr = c.iterator();
```

Iterator object can be created by calling *iterator()* method present in Collection interface.

Limitations of Iterator:

- Only forward direction iterating is possible.
- Replacement and addition of new element is not supported by Iterator.

The RandomAccess Interface

The **RandomAccess** interface contains no members. However, by implementing this interface, a collection signals that it supports efficient random access to its elements. Although a collection

might support random access, it might not do so efficiently. By checking for the **RandomAccess** interface, client code can determine at run time whether a collection is suitable for certain types of random access operations—especially as they apply to large collections. (You can use **instanceof** to determine if a class implements an interface.) **RandomAccess** is implemented

- by **ArrayList** and by the legacy **Vector** class, among others.

Working with Maps

A *map* is an object that stores associations between keys and values, or *key/value pairs*. Given a key, you can find its value. Both keys and values are objects. The keys must be unique, but the values may be duplicated. Some maps can accept a **null** key and **null** values, others cannot.

The Map Interfaces

Because the map interfaces define the character and nature of maps, this discussion of maps begins with them. The following interfaces support maps

Interface	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of Map .
NavigableMap	Extends SortedMap to handle the retrieval of entries based on closest-match searches. (Added by Java SE 6.)
SortedMap	Extends Map so that the keys are maintained in ascending order.

The Map Interface

The **Map** interface maps unique keys to values. A *key* is an object that you use to retrieve a value at a later date. Given a key and a value, you can store the value in a **Map** object. After the value is stored, you can retrieve it by using its key. **Map** is generic and is declared as shown here:

```
interface Map<K, V>
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The methods declared by **Map** are summarized in Table 17-10. Several methods

throw a **ClassCastException** when an object is incompatible with the elements in a map. A

NullPointerException is thrown if an attempt is made to use a **null** object and **null** is not

allowed in the map. An **UnsupportedOperationException** is thrown when an attempt is made to change an unmodifiable map. An **IllegalArgumentException** is thrown if an invalid argument is used. Maps revolve around two basic operations: **get()** and **put()**. To put a value into a map, use **put()**, specifying the key and the value. To obtain a value, call **get()**, passing the key as an argument. The value is returned. As mentioned earlier, although part of the Collections Framework, maps are not, themselves, collections because they do not implement the **Collection** interface. However, you can obtain a collection-view of a map. To do this, you can use the **entrySet()** method. It returns a **Set** that contains the elements in the map. To obtain a collection-view of the keys,

Comparators

Both **TreeSet** and **TreeMap** store elements in sorted order. However, it is the comparator that defines precisely what “sorted order” means. By default, these classes store their elements by using what Java refers to as “natural ordering,” which is usually the ordering that you would expect (A before B, 1 before 2, and so forth). If you want to order elements a different way, then specify a **Comparator** when you construct the set or map. Doing so gives you the ability to govern precisely how elements are stored within sorted collections and maps.

Comparator is a generic interface that has this declaration:

```
interface Comparator<T>
```

Here, **T** specifies the type of objects being compared.

The **Comparator** interface defines two methods: **compare()** and **equals()**. The **compare()** method, shown here, compares two elements for order:

```
int compare(T obj1, T obj2)
```

obj1 and *obj2* are the objects to be compared. This method returns zero if the objects are equal.

It returns a positive value if *obj1* is greater than *obj2*. Otherwise, a negative value is returned.

The method can throw a **ClassCastException** if the types of the objects are not compatible for comparison. By overriding **compare()**, you can alter the way that objects are ordered. For example, to sort in reverse order, you can create a comparator that reverses the outcome of a comparison.

The **equals()** method, shown here, tests whether an object equals the invoking comparator:

```
boolean equals(Object obj)
```

Here, *obj* is the object to be tested for equality. The method returns **true** if *obj* and the invoking object are both **Comparator** objects and use the same ordering. Otherwise, it returns **false**.

Overriding **equals()** is unnecessary, and most simple comparators will not do so.

Using a Comparator

The following is an example that demonstrates the power of a custom comparator. It implements the **compare()** method for strings that operates in reverse of normal. Thus, it causes a tree set to be stored in reverse order.

```
// Use a custom comparator.
```

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

```
// A reverse comparator for strings.
```

```
class MyComp implements Comparator<String> {

    public int compare(String a, String b) {

        String aStr, bStr;

        aStr = a;

        bStr = b;

        // Reverse the comparison.

        return bStr.compareTo(aStr);

    }

    // No need to override equals.

}

class CompDemo {

    public static void main(String args[]) {

        // Create a tree set.

        TreeSet<String> ts = new TreeSet<String>(new MyComp());

        // Add elements to the tree set.

        ts.add("C");

        ts.add("A");

        ts.add("B");

        ts.add("E");

        ts.add("F");

        ts.add("D");
```

```
// Display the elements.  
  
for(String element : ts)  
  
    System.out.print(element + " ");  
  
    System.out.println();  
  
}  
  
}
```

As the following output shows, the tree is now stored in reverse order:

F E D C B A

The Collection Algorithms

The Collections Framework defines several algorithms that can be applied to collections and maps. These algorithms are defined as static methods within the **Collections** class. They are summarized in Table 17-14. As explained earlier, beginning with JDK 5 all of the algorithms have been retrofitted for generics. Although the generic syntax might seem a bit intimidating at first, the algorithms are as simple to use as they were before generics. It's just that now, they are type safe.

Method	Description
static <T> boolean addAll(Collection <? super T> c, T ... elements)	Inserts the elements specified by <i>elements</i> into the collection specified by <i>c</i> . Returns true if the elements were added and false otherwise.
static <T> Queue<T> asLifoQueue(Deque<T> c)	Returns a last-in, first-out view of <i>c</i> . (Added by Java SE 6.)
static <T> int binarySearch(List<? extends T> list, T value, Comparator<? super T> c)	Searches for <i>value</i> in <i>list</i> ordered according to <i>c</i> . Returns the position of <i>value</i> in <i>list</i> , or a negative value if <i>value</i> is not found.
static <T> int binarySearch(List<? extends Comparable<? super T>> list, T value)	Searches for <i>value</i> in <i>list</i> . The list must be sorted. Returns the position of <i>value</i> in <i>list</i> , or a negative value if <i>value</i> is not found.
static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> t)	Returns a run-time type-safe view of a collection. An attempt to insert an incompatible element will cause a ClassCastException .
static <E> List<E> checkedList(List<E> c, Class<E> t)	Returns a run-time type-safe view of a List . An attempt to insert an incompatible element will cause a ClassCastException .
static <K, V> Map<K, V> checkedMap(Map<K, V> c, Class<K> keyT, Class<V> valueT)	Returns a run-time type-safe view of a Map . An attempt to insert an incompatible element will cause a ClassCastException .
static <E> List<E> checkedSet(Set<E> c, Class<E> t)	Returns a run-time type-safe view of a Set . An attempt to insert an incompatible element will cause a ClassCastException .
static <K, V> SortedMap<K, V> checkedSortedMap(SortedMap<K, V> c, Class<K> keyT, Class<V> valueT)	Returns a run-time type-safe view of a SortedMap . An attempt to insert an incompatible element will cause a ClassCastException .
static <E> SortedSet<E> checkedSortedSet(SortedSet<E> c, Class<E> t)	Returns a run-time type-safe view of a SortedSet . An attempt to insert an incompatible element will cause a ClassCastException .
static <T> void copy(List<? super T> list1, List<? extends T> list2)	Copies the elements of <i>list2</i> to <i>list1</i> .
static boolean disjoint(Collection<?> a, Collection<?> b)	Compares the elements in <i>a</i> to elements in <i>b</i> . Returns true if the two collections contain no common elements (i.e., the collections contain disjoint sets of elements). Otherwise, returns false .
static <T> List<T> emptyList()	Returns an immutable, empty List object of the inferred type.
static <K, V> Map<K, V> emptyMap()	Returns an immutable, empty Map object of the inferred type.
static <T> Set<T> emptySet()	Returns an immutable, empty Set object of the inferred type.
static <T> Enumeration<T> enumeration(Collection<T> c)	Returns an enumeration over <i>c</i> . (See "The Enumeration Interface," later in this chapter.)
static <T> void fill(List<? super T> list, T obj)	Assigns <i>obj</i> to each element of <i>list</i> .

Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**, which occurs when an attempt is made to modify an unmodifiable collection. Other exceptions are possible, depending on the method. One thing to pay special attention to is the set of **checked** methods, such as **checkedCollection()**, which returns what the API documentation refers to as a “dynamically typesafe view” of a collection. This view is a reference to the collection that monitors insertions into the collection for type compatibility at run time. An attempt to insert an incompatible element will cause a **ClassCastException**. Using such a view is especially helpful during debugging because it ensures that the collection always contains valid elements. Related methods include **checkedSet()**, **checkedList()**, **checkedMap()**, and so on. They obtain a type-safe view for the indicated collection.