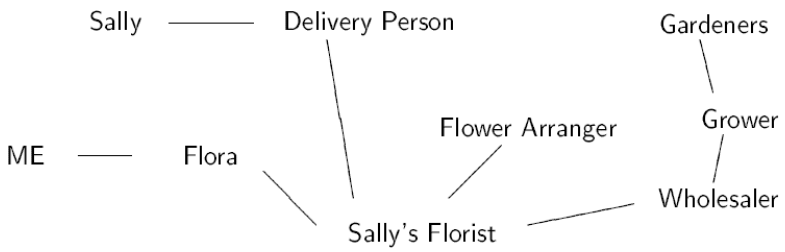
Object-Oriented Thinking

Object-oriented programming is frequently referred to as a new programming paradigm. The word paradigm originally meant example, or model. For example, a paradigm sentence would help you remember how to conjugate a verb in a foreign language. More generally, a model is an example that helps you understand how the world works. For example, the Newtonian model of physics explains why apples fall to the ground. In computer science, a paradigm explains how the elements that go into making a computer program are organized and how they interact with each other. For this reason the first step in understanding Java is appreciating the object-oriented world view.

A Way of Viewing the World

To illustrate the major ideas in object-oriented programming, let us consider how we might go about handling a real-world situation and then ask how we could make the computer more closely model the techniques employed. Suppose I wish to send flowers to a friend who lives in a city many miles away. Let me call my friend Sally. Because of the distance, there is no possibility of my picking the flowers and carrying them to her door myself. Nevertheless, sending her the flowers is an easy enough task; I merely go down to my local florist (who happens to be named Flora), tell her the variety and quantity of flowers I wish to send and give her Sally's address, and I can be assured the flowers will be delivered expediently and automatically.



The community of agents helping me

Agents and Communities

At the risk of belaboring a point, let me emphasize that the mechanism I used to solve my problem was to find an appropriate agent (namely, Flora) and to pass to her a message containing my request. It is the responsibility of Flora to satisfy my request. There is some method-some algorithm or set of operations-used by Flora to do this. I do not need to know the particular method she will use to satisfy my request; indeed, often I do not want to know the details. This information is usually hidden from my inspection.

If I investigated however, I might discover that Flora delivers a slightly different message to another florist in my friend's city. That florist, in turn, perhaps has a subordinate who makes the flower arrangement. The florist then passes the flowers, along with yet another message, to a delivery person, and so on. Earlier, the florist in Sally's city had obtained her flowers from a flower wholesaler who, in turn, had interactions with the flower growers, each of whom had to manage a team of gardeners.

So, our first observation of object-oriented problem solving is that the solution to my problem required the help of many other individuals. Without their help, my problem could not be easily solved. We phrase this in a general fashion as the following:

An object oriented program is structured as a community of interacting agents, called objects. Each object has a role to play. Each object provides a service, or performs an action, that is used by other members of the community.

Messages and Methods

The chain reaction that ultimately resulted in the solution to my program began with my request to Flora. This request lead to other requests, which lead to still more requests, until my owers ultimately reached my friend. We see, therefore, that members of this community interact with each other by making requests. So, our next principle of object- oriented problem solving is the vehicle by which activities are initiated:

Action is initiated in object-oriented programming by the transmission of a message to an agent (an object) responsible for the action. The message encodes the request for an action and is accompanied by any additional information (arguments) needed to carry out the request. The receiver is the object to whom the message is sent. If the receiver accepts the message, it accepts the responsibility to carry out the indicated action. In response to a message, the receiver will perform some method to satisfy the request.

We have noted the important principle of information hiding in regard to message passing-that is, the client sending the request need not know the actual means by which the request will be honored. There is another principle, all too human, that we see is implicit in message passing. If there is a task to perform, the first thought of the client is to find somebody else he or she can ask to do the work. This second reaction often becomes atrophied in many programmers with extensive experience in conventional techniques. Frequently, a difficult hurdle to overcome is the idea in the programmer's mind that he or she must write everything and not use the services of others. An important part of object-oriented programming is the development of reusable components, and an important first step in the use of reusable components is a willingness to trust software written by others.

Information hiding is also an important aspect of programming in conventional languages. In what sense is a message different from, say, a procedure call? In both cases, there is a set of well-defined steps that will be initiated following the request. But, there are two important distinctions.

The first is that in a message there is a designated receiver for that message; the receiver is some object to which the message is sent. In a procedure call, there is no designated receiver.

The second is that the interpretation of the message (that is, the method used to respond to the message) is dependent on the receiver and can vary with different receivers. I can give a message to my wife Elizabeth, for example, and she will understand it and a satisfactory outcome will be produced (that is, flowers will be delivered to my friend). However, the method Elizabeth uses to satisfy the request (in all likelihood, simply passing the request onto Flora) will be different from that used by Flora in response to the same request. If I ask Kenneth, my dentist, to send flowers to my friend, he may not have a method for solving that problem. If he understands the request at all, he will probably issue an appropriate error diagnostic.

Let us move our discussion back to the level of computers and programs. There, the distinction between message passing and procedure calling is that, in message passing, there is a designated receiver, and the interpretation-the selection of a method to execute in response to the message-may vary with different receivers. Usually, the specific receiver for any given message will not be known until run time, so the determination of which method to invoke cannot be made until then. Thus, we say there is late binding between the message (function or procedure name) and the code fragment (method) used to respond to the message. This situation is in contrast to the very early (compile-time or link-time) binding of name to code fragment in conventional procedure calls.

Responsibilities

A fundamental concept in object-oriented programming is to describe behavior in terms of responsibilities. My request for action indicates only the desired outcome (flowers for my friend). Flora is free to pursue any technique that achieves the desired objective and is not hampered by interference on my part.

By discussing a problem in terms of responsibilities we increase the level of abstraction. This permits greater independence between objects, a critical factor in solving complex problems. The entire collection of responsibilities associated with an object is often described by the term protocol.

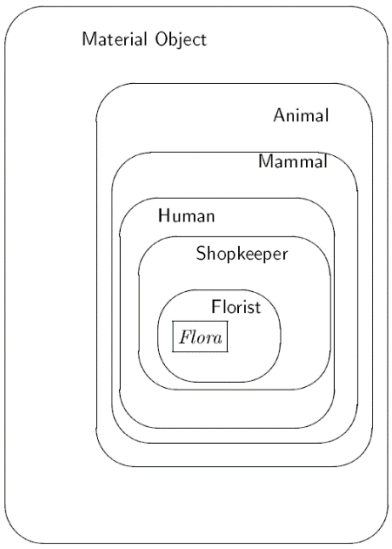
A traditional program often operates by acting on data structures, for example changing fields in an array or record. In contrast, an object oriented program requests data structures (that is, objects) to perform a service. This difference between viewing software in traditional, structured terms and viewing it from an object-oriented perspective can be summarized by a twist on a well-known quote:

Ask not what you can do to your data structures, but rather ask what your data structures can do for you.

Classes and Instances

Although I have only dealt with Flora a few times, I have a rough idea of the behavior I can expect when I go into her shop and present her with my request. I am able to make certain assumptions because I have information about florists in general, and I expect that Flora, being an instance of this category, will fit the general pattern. We can use the term Florist to represent the category (or class) of all florists. Let us incorporate these notions into our next principle of object-oriented programming:

All objects are instances of a class. The method invoked by an object in response to a message is determined by the class of the receiver. All objects of a given class use the same method in response to similar messages.

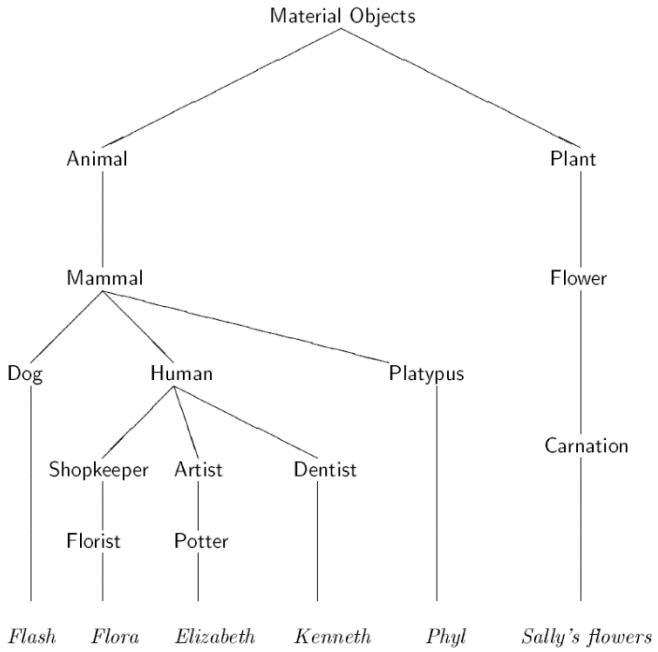


The categories surrounding Flora.

Class Hierarchies-Inheritance

I have more information about Flora-not necessarily because she is a florist but because she is a shopkeeper. I know, for example, that I probably will be asked for money as part of the transaction, and that in return for payment I will be given a receipt. These actions are true of grocers, stationers, and other shopkeepers. Since the category Florist is a more specialized form of the category Shopkeeper, any knowledge I have of Shopkeepers is also true of Florists and hence of Flora.

One way to think about how I have organized my knowledge of Flora is in terms of a hierarchy of categories (see Figure 1.2). Flora is a Florist, but Florist is a specialized form of Shopkeeper. Furthermore, a Shopkeeper is also a Human; so I know, for example, that Flora is probably bipedal. A Human is a Mammal (therefore they nurse their young and have hair), and a Mammal is an Animal (therefore it breathes oxygen), and an Animal is a Material Object (therefore it has mass and weight). Thus, quite a lot of knowledge that I have that is applicable to Flora is not directly associated with her, or even with her category Florist.



A class hierarchy for various material objects.

The principle that knowledge of a more general category is also applicable to a more specific category is called inheritance. We say that the class Florist will inherit attributes of the class (or category) Shopkeeper.

There is an alternative graphical technique often used to illustrate this relationship, particularly when there are many individuals with differing lineage's. This technique shows classes listed in a hierarchical tree-like structure, with more abstract classes (such as Material Object or Animal) listed near the top of the tree, and more specific classes, and finally individuals, are listed near the bottom. Above Fig shows this class hierarchy for Flora. This same hierarchy also includes Elizabeth, my dog Flash, Phyl the platypus who lives at the zoo, and the flowers I am sending to my friend.

Information that I possess about Flora because she is an instance of class Human is also applicable to my wife Elizabeth, for example. Information that I have about her because she is a Mammal is applicable to Flash as well. Information about all members of Material Object is equally applicable to Flora and to her flowers. We capture this in the idea of inheritance:

Classes can be organized into a hierarchical inheritance structure. A child class (or subclass) will inherit attributes from a parent class higher in the tree. An abstract parent class is a class (such as Mammal) for which there are no direct instances; it is used only to create subclasses.

Method Binding, Overriding, and Exceptions

Phyl the platypus presents a problem for our simple organizing structure. I know that mammals give birth to live children, and Phyl is certainly a Mammal, yet Phyl (or rather his mate Phyllis) lays eggs. To accommodate this, we need to find a technique to encode exceptions to a general rule.

We do this by decreeing that information contained in a subclass can override information inherited from a parent class. Most often, implementations of this approach takes the form of a method in a subclass having the same name as a method in the parent class, combined with a rule for how the search for a method to match a specific message is conducted:

The search for a method to invoke in response to a given message begins with the class of the receiver. If no appropriate method is found, the search is conducted in the parent class of this class. The search continues up the parent class chain until either a method is found or the parent class chain is exhausted. In the former case the method is executed; in the latter case, an error message is issued. If methods with the same name can be found higher in the class hierarchy, the method executed is said to override the inherited behavior.

Even if the compiler cannot determine which method will be invoked at run time, in many object-oriented languages, such as Java, it can determine whether there will be an appropriate method and issue an error message as a compile-time error diagnostic rather than as a run-time message.

That my wife Elizabeth and my florist Flora will respond to my message by different methods is an example of one form of polymorphism. We will discuss this important part of object-oriented programming in Chapter 12. As explained, that I do not, and need not, know exactly what method Flora will use to honor my message is an example of information hiding.

Summary of Object-Oriented Concepts

Alan Kay, considered by some to be the father of object-oriented programming, identified the following characteristics as fundamental to OOP [Kay 1993]:

1. Everything is an object.

2. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving messages. A message is a request for action bundled with whatever arguments may be necessary to complete the task.

3. Each object has its own memory, which consists of other objects.

4. Every object is an instance of a class. A class simply represents a grouping of similar objects, such as integers or lists.

5. The class is the repository for behavior associated with an object. That is, all objects that are instances of the same class can perform the same actions.

6. Classes are organized into a singly rooted tree structure, called the inheritance hierarchy. Memory and behavior associated with instances of a class are automatically available to any class associated with a descendant in this tree structure.

Overview of Java

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

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

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

Application

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

Types of Java Applications

There are mainly 4 types of applications that can be created using Java programming:

1) Standalone Application

Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.

2) Web Application

An application that runs on the server side and creates a dynamic page is called a web application. Currently, Servlet, JSP, Struts, Spring, Hibernate, JSF, etc. technologies are used for creating web applications in Java.

3) Enterprise Application

An application that is distributed in nature, such as banking applications, etc. is called enterprise application. It has advantages of the high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.

4) Mobile Application

An application which is created for mobile devices is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

Java Platforms / Editions

There are 4 platforms or editions of Java:

1) Java SE (Java Standard Edition)

It is a Java programming platform. It includes Java programming APIs such as java.lang, java.io, java.net, java.util, java.sql, java.math etc. It includes core topics like OOPs, String, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.

2) Java EE (Java Enterprise Edition)
It is an enterprise platform which is mainly used to develop web and enterprise applications. It is built on the top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA, etc.

3) Java ME (Java Micro Edition)
It is a micro platform which is mainly used to develop mobile applications.

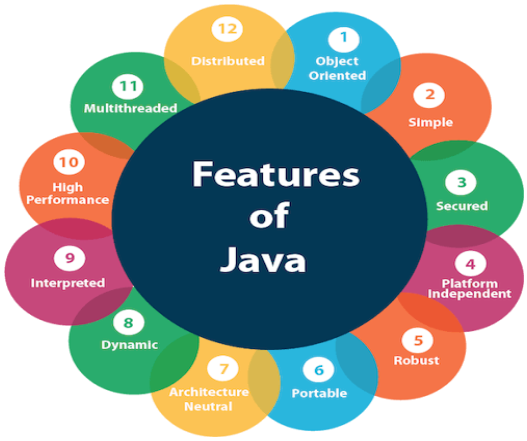
4) JavaFX
It is used to develop rich internet applications. It uses a light-weight user interface API.

Java buzzwords

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

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

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



Simple

Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

Object-oriented

Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

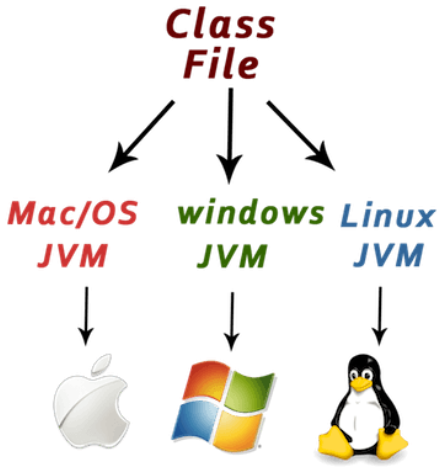
Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance

4. Polymorphism
5. Abstraction
6. Encapsulation

Platform Independent

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



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

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

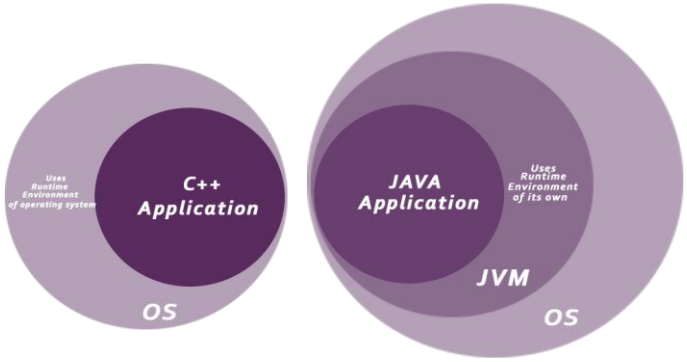
1. Runtime Environment
2. API(Application Programming Interface)

Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).

Secured

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

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**



- **Classloader:** Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.

- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

Robust

Robust simply means strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Architecture-neutral

Java is architecture neutral because there are no implementation dependent features, for example, the size of primitive types is fixed.

In C programming, int data type occupies 2 bytes of memory for 32-bit architecture and 4 bytes of memory for 64-bit architecture. However, it occupies 4 bytes of memory for both 32 and 64-bit architectures in Java.

Portable

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

High-performance

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

Distributed

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

Multi-threaded

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

Dynamic

Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

Java supports dynamic compilation and automatic memory management (garbage collection).

Data Types in Java

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

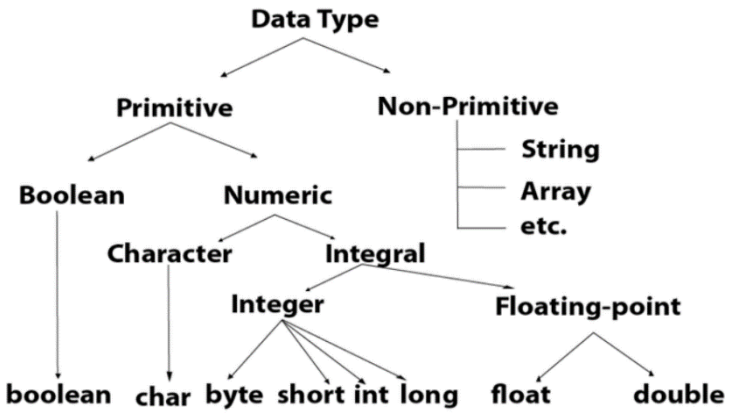
1. **Primitive data types:** The primitive data types include boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

Java Primitive Data Types

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language.

There are 8 types of primitive data types:

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



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

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 isan 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 - 2,147,483,648and 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 if 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 - 9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and 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

JAVA Unit-1

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'

Why char uses 2 byte in java and what is \u0000 ?

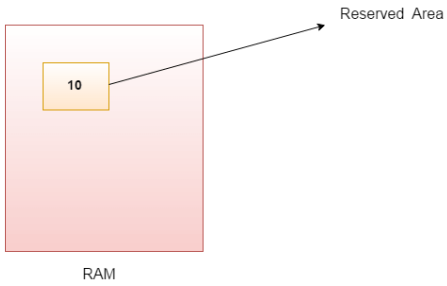
It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system.

Java Variables

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

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

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




```
int data=50;//Here data is variable
```

Types of Variables

There are three types of variables in java:

- local variable
- instance variable
- static variable

1) Local Variable

A variable declared inside the body of the method is called local variable. You can use this variable only within that method and the other methods in the class aren't even aware that the variable exists.

A local variable cannot be defined with "static" keyword.

--

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 static variable and share among all the instances of the class. Memory allocation for static variable happens only once when the class is loaded in the memory.

Ex:

```
1. class A{
2. int data=50;//instance variable
3. static int m=100;//static variable
4. void method(){
5. int n=90;//local variable
6. }
7. }//end of class
```

Add Two Numbers

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

Widening

```
1. class Simple{
2. public static void main(String[] args){
3. int a=10;
4. float f=a;
5. System.out.println(a);
6. System.out.println(f);
7. }}
```

Narrowing (Typecasting)

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

```
7. System.out.println(a);
8. }}
```

Overflow

```
1. class Simple{
2. public static void main(String[] args){
3. //Overflow
4. int a=130;
5. byte b=(byte)a;
6. System.out.println(a);
7. System.out.println(b);
8. }}
```

Adding Lower Type

```
1. class Simple{
2. public static void main(String[] args){
3. byte a=10;
4. byte b=10;
5. //byte c=a+b;//Compile Time Error: because a+b=20 will be int
6. byte c=(byte)(a+b);
7. System.out.println(c);
8. }}
```

Java Arrays

Normally, an array is a collection of similar type of elements which have a contiguous memory location.

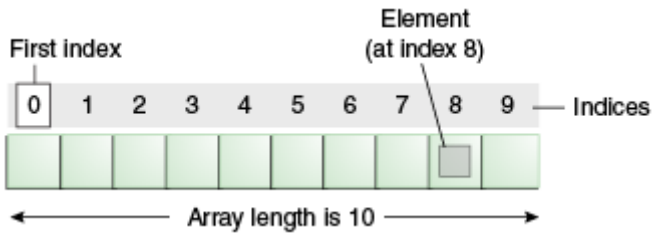
Java array is an object which contains elements of a similar data type. Additionally, The elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array.

Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.

In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimentional or multidimentional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Advantages

- **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
- **Random access:** We can get any data located at an index position.

Disadvantages

Size Limit: We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

There are two types of array.

- Single Dimensional Array
- Multidimensional Array

Single Dimensional Array in Java

Syntax

dataType[] arr; (or)
dataType []arr; (or)
dataType arr[];

Instantiation of an Array in Java

arrayRefVar=**new** datatype[size];

Examples: 1

```
1.  //Java Program to illustrate how to declare, instantiate, initialize and traverse the Java array.
2.  class Testarray{
3.  public static void main(String args[]){
4.  int a[]=new int[5];//declaration and instantiation
5.  a[0]=10;//initialization
6.  a[1]=20;
7.  a[2]=70;
8.  a[3]=40;
9.  a[4]=50;
10. //traversing array
11. for(int i=0;i<a.length;i++)//length is the property of array
12. System.out.println(a[i]);
13. }}
```

Examples: 2

```
1.  //Java Program to illustrate the use of declaration, instantiation and initialization of Java array in a single line
2.  class Testarray1{
3.  public static void main(String args[]){
4.  int a[]={33,3,4,5};//declaration, instantiation and initialization
5.  //printing array
6.  for(int i=0;i<a.length;i++)//length is the property of array
7.  System.out.println(a[i]);
8.  }}
```

Examples: 3

```
1. //Java Program to print the array elements using for-each loop
2. class Testarray1{
3. public static void main(String args[]){
4. int arr[]={33,3,4,5};
5. //printing array using for-each loop
6. for(int i:arr)
7. System.out.println(i);
8. }}
```

Examples: 4

```
1. //Java Program to demonstrate the way of passing an array to method.
2. class Testarray2{
3. //creating a method which receives an array as a parameter
4. static void min(int arr[]){
5. int min=arr[0];
6. for(int i=1;i<arr.length;i++)
7.  if(min>arr[i])
8.   min=arr[i];
9.
10. System.out.println(min);
11. }
12.
13. public static void main(String args[]){
14. int a[]={33,3,4,5};//declaring and initializing an array
15. min(a);//passing array to method
16. }}
```

Examples: 5

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

```
1. //Java Program to demonstrate the way of passing an anonymous array to method.
2. public class TestAnonymousArray{
3. //creating a method which receives an array as a parameter
4. static void printArray(int arr[]){
5. for(int i=0;i<arr.length;i++)
6. System.out.println(arr[i]);
7. }
8.
9. public static void main(String args[]){
10. printArray(new int[]{10,22,44,66});//passing anonymous array to method
11. }}
```

Examples: 6

```
1. //Java Program to return an array from the method
2. class TestReturnArray{
3. //creating method which returns an array
```

```
4. static int[] get(){
5. return new int[]{10,30,50,90,60};
6. }
7.  public static void main(String args[]){
8. //calling method which returns an array
9. int arr[]=get();
10. //printing the values of an array
11. for(int i=0;i<arr.length;i++)
12. System.out.println(arr[i]);
13. }}
```

Examples: 7

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array in negative, equal to the array size or greater than the array size while traversing the array.

```
1. //Java Program to demonstrate the case of ArrayIndexOutOfBoundsException in a Java Array.
2. public class TestArrayException{
3. public static void main(String args[]){
4. int arr[]={50,60,70,80};
5. for(int i=0;i<=arr.length;i++){
6. System.out.println(arr[i]);
7. }
8. }}
```

Multidimensional Array in Java

In such case, data is stored in row and column based index (also known as matrix form).

Syntax

dataType[][] arrayRefVar; (or)
dataType [][]arrayRefVar; (or)
dataType arrayRefVar[][]; (or)
dataType []arrayRefVar[];

instantiate Multidimensional Array

```
int[][] arr=new int[3][3];//3 row and 3 column

arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

Examples: 1

```
1. //Java Program to illustrate the use of multidimensional 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();
}
}}
```

Jagged Array in Java

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

Examples:

```
//Java Program to illustrate the jagged array
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
        }
    }
}
```

Examples: 3

```
//Java Program to get the class name of array in Java
class Testarray4{
public static void main(String args[]){
//declaration and initialization of array
int arr[]={4,4,5};
//getting the class name of Java array
```

```
Class c=arr.getClass();
String name=c.getName();
//printing the class name of Java array
System.out.println(name);

}}
```

Copying an Array in Java
Examples: 4

```
//Java Program to copy a source array into a destination array in Java
class TestArrayCopyDemo {
    public static void main(String[] args) {
        //declaring a source array
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd' };
        //declaring a destination array
        char[] copyTo = new char[7];
        //copying array using System.arraycopy() method
        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        //printing the destination array
        System.out.println(String.valueOf(copyTo));
    }
}
```

Cloning an Array in Java

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

Examples: 5

```
//Java Program to clone the array
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
System.out.println("Printing original array:");
for(int i:arr)
System.out.println(i);
System.out.println("Printing clone of the array:");
int carr[]=arr.clone();
for(int i:carr)
System.out.println(i);
System.out.println("Are both equal?");
System.out.println(arr==carr);
}}
```

Examples: 6

```
//Java Program to demonstrate the addition of two matrices in Java
class Testarray5{
```

```
3. public static void main(String args[]){
4. //creating two matrices
5. int a[][]={{1,3,4},{3,4,5}};
6. int b[][]={{1,3,4},{3,4,5}};
7. //creating another matrix to store the sum of two matrices
8. int c[][]=new int[2][3];
9. //adding and printing addition of 2 matrices
10. for(int i=0;i<2;i++){
11. for(int j=0;j<3;j++){
12. c[i][j]=a[i][j]+b[i][j];
13. System.out.print(c[i][j]+" ");
14. }
15. System.out.println();//new line
16. }
17. }}
```

Examples: 7

```
1. //Java Program to multiply two matrices
2. public class MatrixMultiplicationExample{
3. public static void main(String args[]){
4. //creating two matrices
5. int a[][]={{1,1,1},{2,2,2},{3,3,3}};
6. int b[][]={{1,1,1},{2,2,2},{3,3,3}};
7. //creating another matrix to store the multiplication of two matrices
8. int c[][]=new int[3][3];  //3 rows and 3 columns
9. //multiplying and printing multiplication of 2 matrices
10. for(int i=0;i<3;i++){
11. for(int j=0;j<3;j++){
12. c[i][j]=0;
13. for(int k=0;k<3;k++)
14. {
15. c[i][j]+=a[i][k]*b[k][j];
16. }//end of k loop
17. System.out.print(c[i][j]+" ");  //printing matrix element
18. }//end of j loop
19. System.out.println();//new line
20. }
21. }}
```

Operators in java

Operator in java is a symbol that is used to perform operations. For example: +, -, *, / etc.
There are many types of operators in java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,

- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Operator Type	Category	Precedence
Unary	postfix	*expr*++ *expr*--
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Java 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

Example 1: ++ and --

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int x=10;
4. System.out.println(x++);//10 (11)
5. System.out.println(++x);//12
6. System.out.println(x--);//12 (11)
7. System.out.println(--x);//10
8. }}
```

Example 2: ++ and --

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=10;
5. System.out.println(a++ + ++a);//10+12=22
6. System.out.println(b++ + b++);//10+11=21
7.
8. }}
```

Example: ~ and !

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

Java Arithmetic Operators

Java arithmatic operators are used to perform addition, subtraction, multiplication, and division. They act as basic mathematical operations.

Example :1

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. System.out.println(a+b);//15
6. System.out.println(a-b);//5
7. System.out.println(a*b);//50
8. System.out.println(a/b);//2
```

```
9. System.out.println(a%b);//0
10. }}
```

Example :2

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

Java 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.

Example

```
1. class OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10<<2);//10*2^2=10*4=40
4. System.out.println(10<<3);//10*2^3=10*8=80
5. System.out.println(20<<2);//20*2^2=20*4=80
6. System.out.println(15<<4);//15*2^4=15*16=240
7. }}
```

Java 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.

Example

```
1. class OperatorExample{
2. public static void main(String args[]){
3. System.out.println(10>>2);//10/2^2=10/4=2
4. System.out.println(20>>2);//20/2^2=20/4=5
5. System.out.println(20>>3);//20/2^3=20/8=2
6. }}
```

Java Shift Operator Example: >> vs >>>

Example: >> vs >>>

```
1. class OperatorExample{
2. public static void main(String args[]){
3.    //For positive number, >> and >>> works same
4.    System.out.println(20>>2);
5.    System.out.println(20>>>2);
6.    //For negative number, >>> changes parity bit (MSB) to 0
7.    System.out.println(-20>>2);
8.    System.out.println(-20>>>2);
9. }}
```

Java AND Operator: Logical && and Bitwise &

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

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

Example: 1

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int c=20;
6. System.out.println(a<b&&a<c);//false && true = false
7. System.out.println(a<b&a<c);//false & true = false
8. }}
```

Example: 2

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int c=20;
6. System.out.println(a<b&&a++<c);//false && true = false
7. System.out.println(a);//10 because second condition is not checked
8. System.out.println(a<b&a++<c);//false && true = false
9. System.out.println(a);//11 because second condition is checked
10. }}
```

Java OR Operator Example: Logical || and Bitwise |

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

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

Example:

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int c=20;
6. System.out.println(a>b||a<c);//true || true = true
7. System.out.println(a>b|a<c);//true | true = true
8. //|| vs |
9. System.out.println(a>b||a++<c);//true || true = true
10. System.out.println(a);//10 because second condition is not checked
11. System.out.println(a>b|a++<c);//true | true = true
12. System.out.println(a);//11 because second condition is checked
13. }}
```

Java Ternary Operator

Java Ternary operator is used as one liner replacement for if-then-else statement and used a lot in java programming. it is the only conditional operator which takes three operands.

Example: 1

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=2;
4. int b=5;
5. int min=(a<b)?a:b;
6. System.out.println(min);
7. }}
```

Example: 2

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=5;
5. int min=(a<b)?a:b;
6. System.out.println(min);
7. }}
```

Java Assignment Operator

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

Example: 1

```
1. class OperatorExample{
2. public static void main(String args[]){
3. int a=10;
4. int b=20;
5. a+=4;//a=a+4 (a=10+4)
6. b-=4;//b=b-4 (b=20-4)
7. System.out.println(a);
8. System.out.println(b);
9. }}
```

Example: 2

```
1. class OperatorExample{
2. public static void main(String[] args){
3. int a=10;
4. a+=3;//10+3
5. System.out.println(a);
6. a-=4;//13-4
7. System.out.println(a);
8. a*=2;//9*2
9. System.out.println(a);
10. a/=2;//18/2
11. System.out.println(a);
12. }}
```

Example: 3

```
1. class OperatorExample{
2. public static void main(String args[]){
3. short a=10;
4. short b=10;
5. a=(short)(a+b);//20 which is int now converted to short
6. System.out.println(a);
7. }}
```

Java Expressions

Expressions consist of variables, operators, literals and method calls that evaluates to a single value.

Example

```
int score;
score = 90;
```

Here, score = 90 is an expression that returns int.

```
Double a = 2.2, b = 3.4, result;
result = a + b - 3.4;
```

Here, a + b - 3.4 is an expression.

```
if (number1 == number2)
    System.out.println("Number 1 is larger than number 2");
```

Here, number1 == number2 is an expression that returns Boolean. Similarly, "Number 1 is larger than number 2" is a string expression.

control statements

Java If-else Statement

The Java *if statement* is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in java.

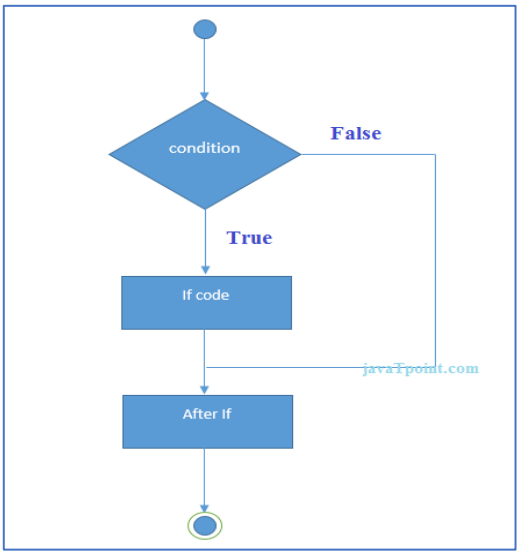
- if statement
- if-else statement
- if-else-if ladder
- nested if statement

Java if Statement

The Java if statement tests the condition. It executes the *if block* if condition is true.

Syntax:

```
1. if(condition){
2. //code to be executed
3. }
```



Example

```
1. //Java Program to demonstate the use of if statement.
2. public class IfExample {
3. public static void main(String[] args) {
4.    //defining an 'age' variable
5.    int age=20;
6.    //checking the age
7.    if(age>18){
8.       System.out.print("Age is greater than 18");
9.    }
```

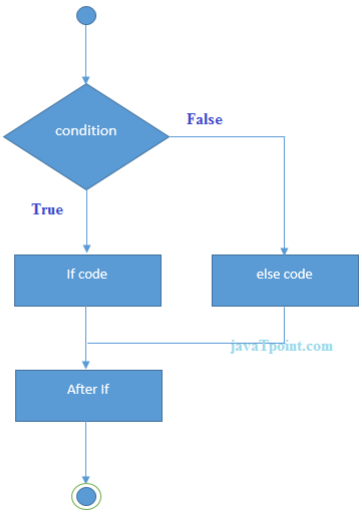
```
}
}
```

Java if-else Statement

The Java if-else statement also tests the condition. It executes the *if block* if condition is true otherwise *else block* is executed.

Syntax:

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```



Example

```
//A Java Program to demonstrate the use of if-else statement.
//It is a program of odd and even number.
public class IfElseExample {
public static void main(String[] args) {
    //defining a variable
    int number=13;
    //Check if the number is divisible by 2 or not
    if(number%2==0){
        System.out.println("even number");
    }else{
        System.out.println("odd number");
    }
}
}
```

Example

```
public class LeapYearExample {
public static void main(String[] args) {
    int year=2020;
    if(((year % 4 ==0) && (year % 100 !=0)) || (year % 400==0)){
        System.out.println("LEAP YEAR");
    }
    else{
        System.out.println("COMMON YEAR");
    }
}
}
```

Java if-else-if ladder Statement

The if-else-if ladder statement executes one condition from multiple statements.

Syntax:

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

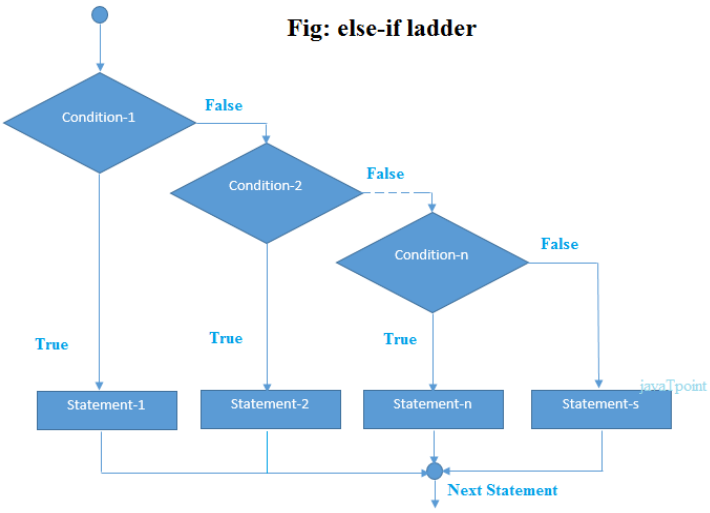


Fig: else-if ladder

Example

```
//Java Program to demonstrate the use of If else-if ladder.
//It is a program of grading system for fail, D grade, C grade, B grade, A grade and A+.
public class IfElseIfExample {
public static void main(String[] args) {
    int marks=65;
    if(marks<50){
        System.out.println("fail");
    }
    else if(marks>=50 && marks<60){
        System.out.println("D grade");
    }
    else if(marks>=60 && marks<70){
        System.out.println("C grade");
    }
    else if(marks>=70 && marks<80){
        System.out.println("B grade");
    }
    else if(marks>=80 && marks<90){
        System.out.println("A grade");
    }else if(marks>=90 && marks<100){
        System.out.println("A+ grade");
    }else{
        System.out.println("Invalid!");
    }
}
}
```

Example

```
public class PositiveNegativeExample {
public static void main(String[] args) {
    int number=-13;
    if(number>0){
    System.out.println("POSITIVE");
    }else if(number<0){
    System.out.println("NEGATIVE");
    }else{
    System.out.println("ZERO");
   }
}
}
```

Java Nested if statement

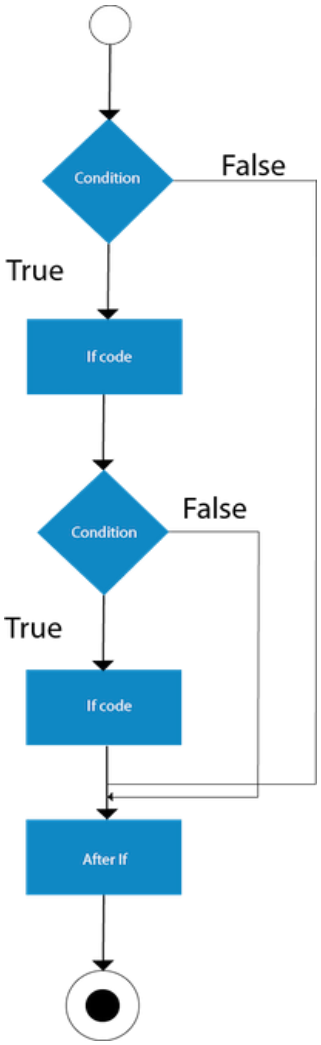
The nested if statement represents the *if block within another if block*. Here, the inner if block condition executes only when outer if block condition is true.

Syntax:

```
if(condition){
    //code to be executed
        if(condition){
          //code to be executed
    }
}
```

Example

```
//Java Program to demonstrate the use of Nested If Statement.
public class JavaNestedIfExample {
public static void main(String[] args) {
   //Creating two variables for age and weight
   int age=20;
   int weight=80;
   //applying condition on age and weight
   if(age>=18){
      if(weight>50){
         System.out.println("You are eligible to donate blood");
      }
   }
}}
```



Example

```
//Java Program to demonstrate the use of Nested If Statement.
public class JavaNestedIfExample2 {
public static void main(String[] args) {
   //Creating two variables for age and weight
   int age=25;
   int weight=48;
   //applying condition on age and weight
   if(age>=18){
      if(weight>50){
         System.out.println("You are eligible to donate blood");
      } else{
         System.out.println("You are not eligible to donate blood");
      }
   } else{
     System.out.println("Age must be greater than 18");
   }
} }
```

Java Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement.

In other words, the switch statement tests the equality of a variable against multiple values.

Points to Remember

- There can be *one or N number of case values* for a switch expression.
- The case value must be of switch expression type only. The case value must be *literal or constant*. It doesn't allow variables.
- The case values must be *unique*. In case of duplicate value, it renders compile-time error.
- The Java switch expression must be of *byte, short, int, long (with its Wrapper type), enums and string*.
- Each case statement can have a *break statement* which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
- The case value can have a *default label* which is optional.

Syntax:

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......
default:
 code to be executed if all cases are not matched;
}
```

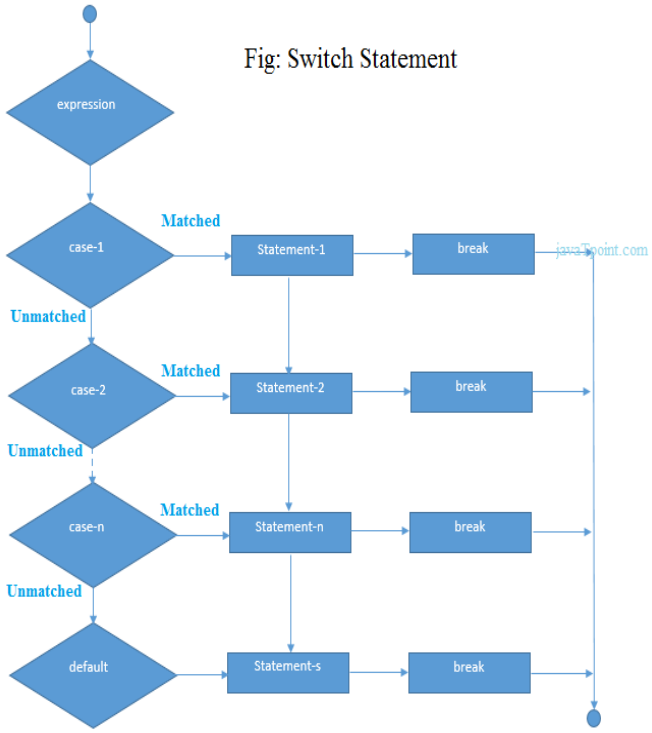


Fig: Switch Statement

Example

```
public class SwitchExample {
public static void main(String[] args) {
    //Declaring a variable for switch expression
    int number=20;
    //Switch expression
    switch(number){
    //Case statements
    case 10: System.out.println("10");
    break;
    case 20: System.out.println("20");
    break;
    case 30: System.out.println("30");
    break;
    //Default case statement
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

Example

```
//Java Program to demonstrate the example of Switch statement
//where we are printing month name for the given number
public class SwitchMonthExample {
public static void main(String[] args) {
    //Specifying month number
    int month=7;
    String monthString="";
    //Switch statement
    switch(month){
    //case statements within the switch block
```

```
    case 1: monthString="1 - January";
    break;
    case 2: monthString="2 - February";
    break;
    case 3: monthString="3 - March";
    break;
    case 4: monthString="4 - April";
    break;
    case 5: monthString="5 - May";
    break;
    case 6: monthString="6 - June";
    break;
    case 7: monthString="7 - July";
    break;
    case 8: monthString="8 - August";
    break;
    case 9: monthString="9 - September";
    break;
    case 10: monthString="10 - October";
    break;
    case 11: monthString="11 - November";
    break;
    case 12: monthString="12 - December";
    break;
    default:System.out.println("Invalid Month!");
    }
    //Printing month of the given number
    System.out.println(monthString);
}
}
```

Example

```
public class SwitchVowelExample {
public static void main(String[] args) {
    char ch='O';
    switch(ch)
    {
        case 'a':
            System.out.println("Vowel");
            break;
        case 'e':
            System.out.println("Vowel");
            break;
        case 'i':
            System.out.println("Vowel");
            break;
        case 'o':
            System.out.println("Vowel");
            break;
        case 'u':
            System.out.println("Vowel");
            break;
        case 'A':
            System.out.println("Vowel");
            break;
        case 'E':
            System.out.println("Vowel");
            break;
        case 'I':
            System.out.println("Vowel");
```

```
            break;
        case 'O':
            System.out.println("Vowel");
            break;
        case 'U':
            System.out.println("Vowel");
            break;
        default:
            System.out.println("Consonant");
    }
}
}
```

Java Switch Statement is fall-through

The Java switch statement is fall-through. It means it executes all statements after the first match if a break statement is not present

Example

```
//Java Switch Example where we are omitting the break statement
public class SwitchExample2 {
public static void main(String[] args) {
    int number=20;
    //switch expression with int value
    switch(number){
    //switch cases without break statements
    case 10: System.out.println("10");
    case 20: System.out.println("20");
    case 30: System.out.println("30");
    default:System.out.println("Not in 10, 20 or 30");
    }
}
}
```

Java Switch Statement with String

Java allows us to use strings in switch expression since Java SE 7. The case statement should be string literal.

Example

```
//Java Program to demonstrate the use of Java Switch statement with String
public class SwitchStringExample {
public static void main(String[] args) {
    //Declaring String variable
    String levelString="Expert";
    int level=0;
    //Using String in Switch expression
    switch(levelString){
    //Using String Literal in Switch case
    case "Beginner": level=1;
    break;
    case "Intermediate": level=2;
    break;
    case "Expert": level=3;
    break;
    default: level=0;
    break;
    }
    System.out.println("Your Level is: "+level);
}
}
```

Java Nested Switch Statement

We can use switch statement inside other switch statement in Java. It is known as nested switch statement.

Example

```
1.  //Java Program to demonstrate the use of Java Nested Switch
2.  public class NestedSwitchExample {
3.     public static void main(String args[])
4.      {  //C - CSE, E - ECE, M - Mechanical
5.        char branch = 'C';
6.        int collegeYear = 4;
7.        switch( collegeYear )
8.        {
9.           case 1:
10.             System.out.println("English, Maths, Science");
11.             break;
12.          case 2:
13.             switch( branch )
14.             {
15.                case 'C':
16.                   System.out.println("Operating System, Java, Data Structure");
17.                   break;
18.                case 'E':
19.                   System.out.println("Micro processors, Logic switching theory");
20.                   break;
21.                case 'M':
22.                   System.out.println("Drawing, Manufacturing Machines");
23.                   break;
24.             }
25.             break;
26.          case 3:
27.             switch( branch )
28.             {
29.                case 'C':
30.                   System.out.println("Computer Organization, MultiMedia");
31.                   break;
32.                case 'E':
33.                   System.out.println("Fundamentals of Logic Design, Microelectronics");
34.                   break;
35.                case 'M':
36.                   System.out.println("Internal Combustion Engines, Mechanical Vibration");
37.                   break;
38.             }
39.             break;
40.          case 4:
41.             switch( branch )
42.             {
43.                case 'C':
44.                   System.out.println("Data Communication and Networks, MultiMedia");
45.                   break;
46.                case 'E':
47.                   System.out.println("Embedded System, Image Processing");
48.                   break;
49.                case 'M':
50.                   System.out.println("Production Technology, Thermal Engineering");
51.                   break;
52.             }
53.             break;
54.       }
55.    }
56. }
```

Java Enum in Switch Statement

Java allows us to use enum in switch statement.

Example

```
1. //Java Program to demonstrate the use of Enum in switch statement
2. public class JavaSwitchEnumExample {
3.       public enum Day {  Sun, Mon, Tue, Wed, Thu, Fri, Sat  }
4.       public static void main(String args[])
5.       {
6.         Day[] DayNow = Day.values();
7.           for (Day Now : DayNow)
8.           {
9.               switch (Now)
10.              {
11.                 case Sun:
12.                    System.out.println("Sunday");
13.                    break;
14.                 case Mon:
15.                    System.out.println("Monday");
16.                    break;
17.                 case Tue:
18.                    System.out.println("Tuesday");
19.                    break;
20.                 case Wed:
21.                    System.out.println("Wednesday");
22.                    break;
23.                 case Thu:
24.                    System.out.println("Thursday");
25.                    break;
26.                 case Fri:
27.                    System.out.println("Friday");
28.                    break;
29.                 case Sat:
30.                    System.out.println("Saturday");
31.                    break;
32.             }
33.          }
34.       }
35. }
```

Java Wrapper in Switch Statement

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

Example

```
1. //Java Program to demonstrate the use of Wrapper class in switch statement
2. public class WrapperInSwitchCaseExample {
3.       public static void main(String args[])
4.       {
5.           Integer age = 18;
6.           switch (age)
7.           {
8.              case (16):
9.                 System.out.println("You are under 18.");
10.                break;
11.             case (18):
12.                System.out.println("You are eligible for vote.");
13.                break;
14.             case (65):
15.                System.out.println("You are senior citizen.");
16.                break;
17.             default:
```

```
                System.out.println("Please give the valid age.");
                break;
            }
        }
}
```

Loops in Java

In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true. There are three types of loops in java.

- for loop
- while loop
- do-while loop

Java For Loop

The Java *for loop* is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.
There are three types of for loops in java.

- Simple For Loop
- For-each or Enhanced For Loop
- Labeled For Loop

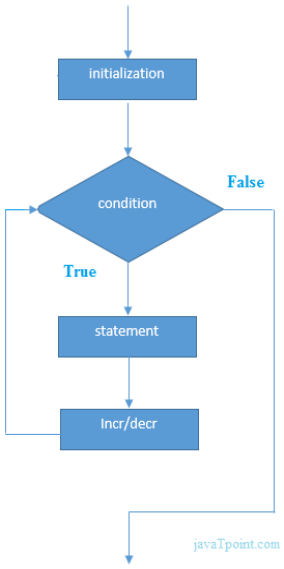
Java Simple For Loop

A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

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.

Syntax:

```
for(initialization;condition;incr/decr){
//statement or code to be executed
}
```



Example

```
//Java Program to demonstrate the example of for loop
//which prints table of 1
public class ForExample {
public static void main(String[] args) {
   //Code of Java for loop
   for(int i=1;i<=10;i++){
      System.out.println(i);
   }
}
}
```

Java Nested For Loop

If we have a for loop inside the another loop, it is known as nested for loop. The inner loop executes completely whenever outer loop executes.

Example:

```
1. public class NestedForExample {
2. public static void main(String[] args) {
3. //loop of i
4. for(int i=1;i<=3;i++){
5. //loop of j
6. for(int j=1;j<=3;j++){
7.         System.out.println(i+" "+j);
8. }//end of i
9. }//end of j
10. }
11. }
```

Example:

```
1. public class PyramidExample {
2. public static void main(String[] args) {
3. for(int i=1;i<=5;i++){
4. for(int j=1;j<=i;j++){
5.         System.out.print("* ");
6. }
7. System.out.println();//new line
8. }
9. }
10. }
```

Example:

```
1. public class PyramidExample2 {
2. public static void main(String[] args) {
3. int term=6;
4. for(int i=1;i<=term;i++){
5. for(int j=term;j>=i;j--){
6.         System.out.print("* ");
7. }
8. System.out.println();//new line
9. }
10. }
11. }
```

Java for-each Loop

The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

It works on elements basis not index. It returns element one by one in the defined variable.

Syntax:

```
1. for(Type var:array){
2. //code to be executed
3. }
```

Example:

```
1. //Java For-each loop example which prints the
2. //elements of the array
3. public class ForEachExample {
4. public static void main(String[] args) {
5.    //Declaring an array
6.    int arr[]={12,23,44,56,78};
7.    //Printing array using for-each loop
8.    for(int i:arr){
9.       System.out.println(i);
10.    }
```

```
}
}
```

Java Labeled For Loop

We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful if we have nested for loop so that we can break/continue specific for loop.

Usually, break and continue keywords breaks/continues the innermost for loop only.

Syntax:

```
labelname:
for(initialization;condition;incr/decr){
//code to be executed
}
```

Example:

```
//A Java program to demonstrate the use of labeled for loop
public class LabeledForExample {
public static void main(String[] args) {
   //Using Label for outer and for loop
   aa:
      for(int i=1;i<=3;i++){
         bb:
            for(int j=1;j<=3;j++){
               if(i==2&&j==2){
                  break aa;
               }
               System.out.println(i+" "+j);
            }
      }
}
}
```

Example:

```
public class LabeledForExample2 {
public static void main(String[] args) {
   aa:
      for(int i=1;i<=3;i++){
         bb:
            for(int j=1;j<=3;j++){
               if(i==2&&j==2){
                  break bb;
               }
               System.out.println(i+" "+j);
            }
      }
}
}
```

Java Infinitive For Loop

If you use two semicolons ;; in the for loop, it will be infinitive for loop.

Syntax:

```
for(;;){
//code to be executed
}
```

Example:

```
//Java program to demonstrate the use of infinite for loop
//which prints an statement
public class ForExample {
```

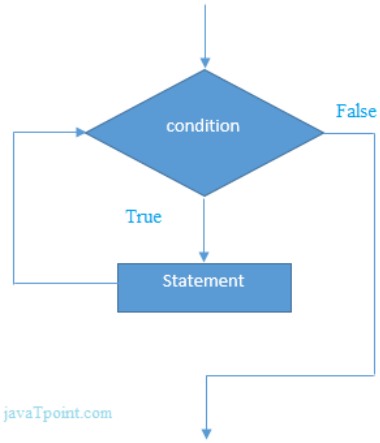
```
4. public static void main(String[] args) {
5.    //Using no condition in for loop
6.    for(;;){
7.       System.out.println("infinitive loop");
8.    }
9. }
10. }
```

Java While Loop

The Java *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

Syntax:

```
1. while(condition){
2. //code to be executed
3. }
```



Example:

```
1. public class WhileExample {
2. public static void main(String[] args) {
3.    int i=1;
4.    while(i<=10){
5.       System.out.println(i);
6.    i++;
7.    }
8. }
9. }
```

Java Infinitive While Loop

If you pass **true** in the while loop, it will be infinitive while loop.

Syntax:

```
1. while(true){
2. //code to be executed
3. }
```

Example:

```
1. public class WhileExample2 {
2. public static void main(String[] args) {
3.    while(true){
4.       System.out.println("infinitive while loop");
5.    }
6. }
7. }
```

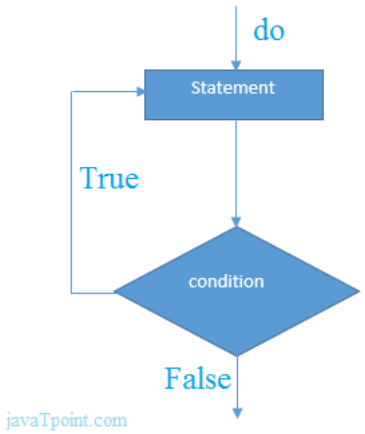
Java do-while Loop

The Java *do-while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The Java *do-while loop* is executed at least once because condition is checked after loop body.

Syntax:

```
do{
//code to be executed
}while(condition);
```



Example:

```
public class DoWhileExample {
public static void main(String[] args) {
    int i=1;
    do{
        System.out.println(i);
    i++;
    }while(i<=10);
}
}
```

Java Infinitive do-while Loop

If you pass **true** in the do-while loop, it will be infinitive do-while loop.

Syntax:

```
 do{
//code to be executed
}while(true);
```

Example:

```
public class DoWhileExample2 {
public static void main(String[] args) {
    do{
        System.out.println("infinitive do while loop");
    }while(true);
}
}
```

Java For Loop vs While Loop vs Do While Loop

Comparison	for loop	while loop	do while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is

			recommended to use the do-while loop.
Syntax	`for(init;condition; incr/decr){ // code to be executed }`	`while(condition){ //code to be executed }`	`do{ //code to be executed }while(condition);`
Example	`//for loop for(int i=1;i<=10;i++){ System.out.println(i); }`	`//while loop int i=1; while(i<=10){ System.out.println(i); i++; }`	`//do-while loop int i=1; do{ System.out.println(i); i++; }while(i<=10);`
Syntax for infinitive loop	`for(;;){ //code to be executed }`	`while(true){ //code to be executed }`	`do{ //code to be executed }while(true);`

Java Break Statement

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

The Java *break* is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

```
1. jump-statement;
2. break;
```

Example:

```
1. //Java Program to demonstrate the use of break statement
2. //inside the for loop.
3. public class BreakExample {
4. public static void main(String[] args) {
5.    //using for loop
6.    for(int i=1;i<=10;i++){
7.       if(i==5){
8.          //breaking the loop
9.          break;
10.      }
11.      System.out.println(i);
12.   }
13. }
14. }
```

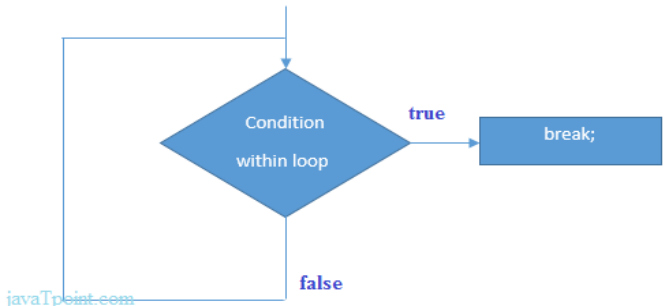


Figure: Flowchart of break statement

Java Break Statement with Inner Loop

It breaks inner loop only if you use break statement inside the inner loop.

Example:

```
1. //Java Program to illustrate the use of break statement
2. //inside an inner loop
3. public class BreakExample2 {
4. public static void main(String[] args) {
5.          //outer loop
6.          for(int i=1;i<=3;i++){
7.                //inner loop
8.                for(int j=1;j<=3;j++){
9.                   if(i==2&&j==2){
10.                     //using break statement inside the inner loop
11.                     break;
12.                  }
13.                  System.out.println(i+" "+j);
14.               }
15.         }
16. }
17. }
```

Java Break Statement with Labeled For Loop

We can use break statement with a label. This feature is introduced since JDK 1.5. So, we can break any loop in Java now whether it is outer loop or inner.

Example:

```
1. //Java Program to illustrate the use of continue statement
2. //with label inside an inner loop to break outer loop
3. public class BreakExample3 {
4. public static void main(String[] args) {
5.          aa:
6.          for(int i=1;i<=3;i++){
7.                bb:
8.                for(int j=1;j<=3;j++){
9.                   if(i==2&&j==2){
10.                     //using break statement with label
11.                     break aa;
12.                  }
13.                  System.out.println(i+" "+j);
14.               }
15.         }
16. }
17. }
```

Java Break Statement in while loop

Example:

```
1. //Java Program to demonstrate the use of break statement
2. //inside the while loop.
3. public class BreakWhileExample {
4. public static void main(String[] args) {
5.    //while loop
6.    int i=1;
7.    while(i<=10){
8.       if(i==5){
9.          //using break statement
10.         i++;
11.         break;//it will break the loop
```

```
        }
        System.out.println(i);
        i++;
    }
}
}
```

Java Break Statement in do-while loop

Example:

```
//Java Program to demonstrate the use of break statement
//inside the Java do-while loop.
public class BreakDoWhileExample {
public static void main(String[] args) {
    //declaring variable
    int i=1;
    //do-while loop
    do{
        if(i==5){
          //using break statement
          i++;
          break;//it will break the loop
        }
        System.out.println(i);
        i++;
    }while(i<=10);
}
}
```

Java Continue Statement

The continue statement is used in loop control structure when you need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

The Java *continue statement* is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only.

We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop.

Syntax:

```
jump-statement;
continue;
```

Example:

```
//Java Program to demonstrate the use of continue statement
//inside the for loop.
public class ContinueExample {
public static void main(String[] args) {
    //for loop
    for(int i=1;i<=10;i++){
        if(i==5){
            //using continue statement
            continue;//it will skip the rest statement
        }
        System.out.println(i);
    }
}
```

```
}
```

Example:

```
//Java Program to illustrate the use of continue statement
//inside an inner loop
public class ContinueExample2 {
public static void main(String[] args) {
        //outer loop
        for(int i=1;i<=3;i++){
            //inner loop
            for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                    //using continue statement inside inner loop
                    continue;
                }
                System.out.println(i+" "+j);
            }
        }
}
}
```

Java Continue Statement with Labeled For Loop

We can use continute statement with a label. This feature is introduced since JDK 1.5. So, we can continue any loop in Java now whether it is outer loop or inner.

Example:

```
//Java Program to illustrate the use of continue statement
//with label inside an inner loop to continue outer loop
public class ContinueExample3 {
public static void main(String[] args) {
        aa:
        for(int i=1;i<=3;i++){
            bb:
            for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                    //using continue statement with label
                    continue aa;
                }
                System.out.println(i+" "+j);
            }
        }
}
}
```

Java Continue Statement in while loop

Example:

```
//Java Program to demonstrate the use of continue statement
//inside the while loop.
public class ContinueWhileExample {
public static void main(String[] args) {
   //while loop
   int i=1;
   while(i<=10){
      if(i==5){
         //using continue statement
         i++;
         continue;//it will skip the rest statement
      }
      System.out.println(i);
```

```
        i++;
    }
}
}
```

Java Continue Statement in do-while loop

Example:

```
//Java Program to demonstrate the use of continue statement
//inside the Java do-while loop.
public class ContinueDoWhileExample {
public static void main(String[] args) {
    //declaring variable
    int i=1;
    //do-while loop
    do{
        if(i==5){
            //using continue statement
             i++;
           continue;//it will skip the rest statement
        }
        System.out.println(i);
        i++;
    }while(i<=10);
}
}
```

Objects and Classes in Java

In object-oriented programming technique, we design a program using objects and classes. An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.

Object in Java

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

An object has 3 characteristics:

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

For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

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

Object Definitions:

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

Class in Java

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

A class in Java can contain:

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

Syntax:

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

Instance variable in Java

A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

Method in Java

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

new keyword in Java

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

Object and Class Example: main within the class

Example: 1

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

Object and Class Example: main outside the class

Example: 2

```
//Java Program to demonstrate having the main method in
//another class
//Creating Student class.
class Student{
 int id;
 String name;
}
```

```
8. //Creating another class TestStudent1 which contains the main method
9. class TestStudent1{
10. public static void main(String args[]){
11.  Student s1=new Student();
12.  System.out.println(s1.id);
13.  System.out.println(s1.name);
14. }
15. }
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

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

1) Object and Class Example: Initialization through reference

Initializing an object means storing data into the object. Let's see a simple example where we are going to initialize the object through a reference variable.

Example: 1

```
1. class Student{
2.  int id;
3.  String name;
4. }
5. class TestStudent2{
6.  public static void main(String args[]){
7.   Student s1=new Student();
8.   s1.id=101;
9.   s1.name="Sonoo";
10.  System.out.println(s1.id+" "+s1.name);//printing members with a white space
11. }
12. }
```

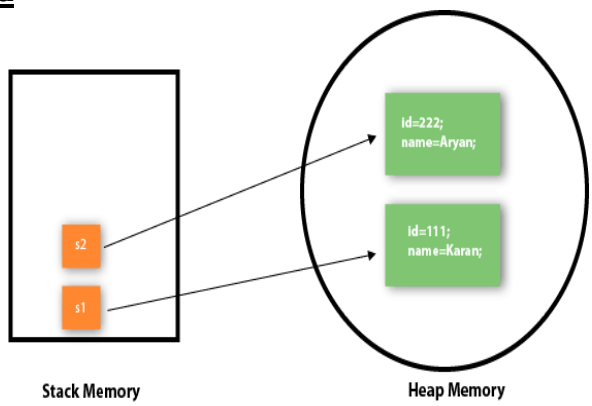
Example: 2

```
1. class Student{
2.  int id;
3.  String name;
4. }
5. class TestStudent3{
6.  public static void main(String args[]){
7.   //Creating objects
8.   Student s1=new Student();
9.   Student s2=new Student();
10.  //Initializing objects
11.  s1.id=101;
12.  s1.name="Sonoo";
13.  s2.id=102;
14.  s2.name="Amit";
15.  //Printing data
16.  System.out.println(s1.id+" "+s1.name);
17.  System.out.println(s2.id+" "+s2.name);
18. }
19. }
```

2) Object and Class Example: Initialization through method

In this example, we are creating the two objects of Student class and initializing the value to these objects by invoking the insertRecord method. Here, we are displaying the state (data) of the objects by invoking the displayInformation() method.

As you can see in the figure, object gets the memory in 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:

```
1.  class Student{
2.   int rollno;
3.   String name;
4.   void insertRecord(int r, String n){
5.    rollno=r;
6.    name=n;
7.   }
8.   void displayInformation(){System.out.println(rollno+" "+name);}
9.  }
10. class TestStudent4{
11.  public static void main(String args[]){
12.   Student s1=new Student();
13.   Student s2=new Student();
14.   s1.insertRecord(111,"Karan");
15.   s2.insertRecord(222,"Aryan");
16.   s1.displayInformation();
17.   s2.displayInformation();
18.  }
19. }
```

3) Object and Class Example: Initialization through a constructor

Example:1

```
1.  class Employee{
2.     int id;
3.     String name;
4.     float salary;
5.     void insert(int i, String n, float s) {
6.        id=i;
7.        name=n;
8.        salary=s;
9.     }
10.    void display(){System.out.println(id+" "+name+" "+salary);}
11. }
12. public class TestEmployee {
13. public static void main(String[] args) {
14.    Employee e1=new Employee();
15.    Employee e2=new Employee();
16.    Employee e3=new Employee();
17.    e1.insert(101,"ajeet",45000);
18.    e2.insert(102,"irfan",25000);
19.    e3.insert(103,"nakul",55000);
20.    e1.display();
21.    e2.display();
22.    e3.display();
23. }
```

```
24. }
```

Example: 2

```
1.  class Rectangle{
2.   int length;
3.   int width;
4.   void insert(int l, int w){
5.    length=l;
6.    width=w;
7.   }
8.   void calculateArea(){System.out.println(length*width);}
9.  }
10. class TestRectangle1{
11.  public static void main(String args[]){
12.   Rectangle r1=new Rectangle();
13.   Rectangle r2=new Rectangle();
14.   r1.insert(11,5);
15.   r2.insert(3,15);
16.   r1.calculateArea();
17.   r2.calculateArea();
18. }
19. }
```

What are the different ways to create an object in Java?

There are many ways to create an object in java. They are:

- By new keyword
- By newInstance() method
- By clone() method
- By deserialization
- By factory method etc.

Anonymous object

Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.

If you have to use an object only once, an anonymous object is a good approach. For example:

```
new Calculation();//anonymous object
```

Calling method through a reference:

```
1. Calculation c=new Calculation();
2. c.fact(5);
```

Calling method through an anonymous object

```
new Calculation().fact(5);
```

Example:

```
1.  class Calculation{
2.   void fact(int  n){
3.    int fact=1;
4.    for(int i=1;i<=n;i++){
5.     fact=fact*i;
6.    }
7.   System.out.println("factorial is "+fact);
8.  }
9.  public static void main(String args[]){
10.  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.
Initialization of primitive variables:

```
int a=10, b=20;
```

Initialization of refernce variables:

```
Rectangle r1=new Rectangle(), r2=new Rectangle();//creating two objects
```

Example: 1

```
1.  //Java Program to illustrate the use of Rectangle class which
2.  //has length and width data members
3.  class Rectangle{
4.   int length;
5.   int width;
6.   void insert(int l,int w){
7.    length=l;
8.    width=w;
9.   }
10.  void calculateArea(){System.out.println(length*width);}
11. }
12. class TestRectangle2{
13.  public static void main(String args[]){
14.   Rectangle r1=new Rectangle(),r2=new Rectangle();//creating two objects
15.   r1.insert(11,5);
16.   r2.insert(3,15);
17.   r1.calculateArea();
18.   r2.calculateArea();
19. }
20. }
```

Example: 2

```
1.  //Java Program to demonstrate the working of a banking-system
2.  //where we deposit and withdraw amount from our account.
3.  //Creating an Account class which has deposit() and withdraw() methods
4.  class Account{
5.  int acc_no;
6.  String name;
7.  float amount;
8.  //Method to initialize object
9.  void insert(int a,String n,float amt){
10. acc_no=a;
11. name=n;
12. amount=amt;
13. }
14. //deposit method
15. void deposit(float amt){
16. amount=amount+amt;
17. System.out.println(amt+" deposited");
18. }
19. //withdraw method
20. void withdraw(float amt){
21. if(amount<amt){
22. System.out.println("Insufficient Balance");
23. }else{
24. amount=amount-amt;
```

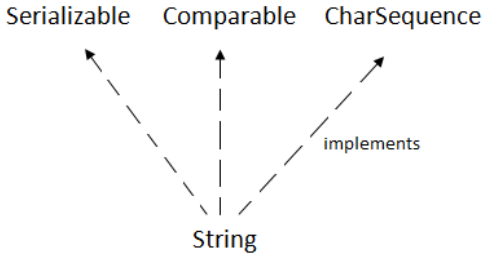
```
25. System.out.println(amt+" withdrawn");
26. }
27. }
28. //method to check the balance of the account
29. void checkBalance(){System.out.println("Balance is: "+amount);}
30. //method to display the values of an object
31. void display(){System.out.println(acc_no+" "+name+" "+amount);}
32. }
33. //Creating a test class to deposit and withdraw amount
34. class TestAccount{
35. public static void main(String[] args){
36. Account a1=new Account();
37. a1.insert(832345,"Ankit",1000);
38. a1.display();
39. a1.checkBalance();
40. a1.deposit(40000);
41. a1.checkBalance();
42. a1.withdraw(15000);
43. a1.checkBalance();
44. }}
```

Java String

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

```
1. char[] ch={'j','a','v','a','t','p','o','i','n','t'};
2. String s=new String(ch);
```

is same as: String s="javatpoint";



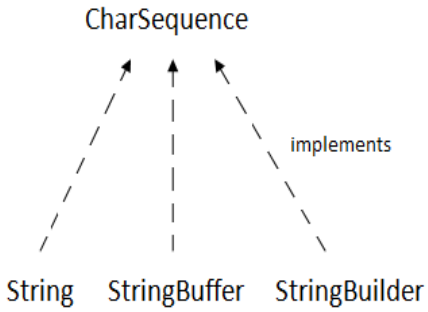
Java String class provides a lot of methods to perform operations on string such as compare(), concat(), equals(), split(), length(), replace(), compareTo(), intern(), substring() etc.

The java.lang.String class implements *Serializable*, *Comparable* and *CharSequence* interfaces.

CharSequence Interface

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in java by using these three classes.

The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.



String in java

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

How to create a string object?

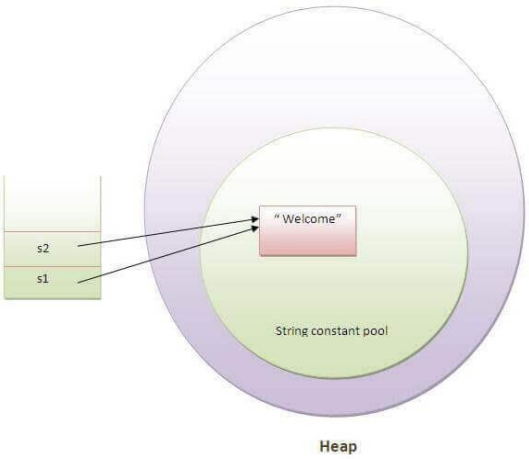
There are two ways to create String object:

1. By string literal
2. By new keyword

1. String Literal

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

```
String s="welcome";
```



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

```
1. String s1="Welcome";
2. String s2="Welcome";//It doesn't create a new instance
```

In the above example, only one object will be created. Firstly, JVM will not find any string object with the value "Welcome" in string constant pool, that is why it will create a new object. After that it will find the string with the value "Welcome" in the pool, it will not create a new object but will return the reference to the same instance.

Note: String objects are stored in a special memory area known as the "string constant pool".

Java uses the concept of String literal because to make Java more memory efficient (because no new objects are created if it exists already in the string constant pool).

2.new keyword

Syntax:

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

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

Example:

```
1. public class StringExample{
2. public static void main(String args[]){
3. String s1="java";//creating string by java string literal
4. char ch[]={'s','t','r','i','n','g','s'};
5. String s2=new String(ch);//converting char array to string
6. String s3=new String("example");//creating java string by new keyword
7. System.out.println(s1);
8. System.out.println(s2);
9. System.out.println(s3);
10. }}
```

Java String class methods

The java.lang.String class provides many useful methods to perform operations on sequence of char values.

Java String charAt()

The **java string charAt()** method returns *a char value at the given index number*.

The index number starts from 0 and goes to n-1, where n is length of the string. It returns **StringIndexOutOfBoundsException** if given index number is greater than or equal to this string length or a negative number.

Example:1

```
1. public class CharAtExample{
2. public static void main(String args[]){
3. String name="CMRTC";
4. char ch=name.charAt(4);//returns the char value at the 4th index
5. System.out.println(ch);
6. }}
```

Example: 2

```
1. public class CharAtExample{
2. public static void main(String args[]){
3. String name="CMRTC";
4. char ch=name.charAt(10);//returns the char value at the 10th index
5. System.out.println(ch);
6. }}
```

Example: 3

```
1. public class CharAtExample3 {
2.    public static void main(String[] args) {
3.    String str = "Welcome to CMRTC CSE";
4.    int strLength = str.length();
5.    // Fetching first character
6.    System.out.println("Character at 0 index is: "+ str.charAt(0));
7.    // The last Character is present at the string length-1 index
8.    System.out.println("Character at last index is: "+ str.charAt(strLength-1));
9.    } }
```

Example: 4

```
1. public class CharAtExample4 {
2.    public static void main(String[] args) {
3.       String str = "Welcome to CMRTC CSE";
4.       for (int i=0; i<=str.length()-1; i++) {
5.          if(i%2!=0) {
6.             System.out.println("Char at "+i+" place "+str.charAt(i));
7.          } } } }
```

Example:5

```
1. public class CharAtExample5 {
2.    public static void main(String[] args) {
3.       String str = "WELCOME to CMRTC CSE";
4.       int count = 0;
5.       for (int i=0; i<=str.length()-1; i++) {
6.          if(str.charAt(i) == 'C') {
7.             count++;
8.          }
9.       }
10.      System.out.println("Frequency of C is: "+count);
11.   } }
```

Java String compareTo()

The **java string compareTo()** method compares the given string with current string lexicographically. It returns positive number, negative number or 0.

It compares strings on the basis of Unicode value of each character in the strings.

If first string is lexicographically greater than second string, it returns positive number (difference of character value). If first string is less than second string lexicographically, it returns negative number and if first string is lexicographically equal to second string, it returns 0.

Example

```
1. public class CompareToExample{
```

```
2. public static void main(String args[]){
3. String s1="hello";
4. String s2="hello";
5. String s3="meklo";
6. String s4="hemlo";
7. String s5="flag";
8. System.out.println(s1.compareTo(s2));//0 because both are equal
9. System.out.println(s1.compareTo(s3));//-5 because "h" is 5 times lower than "m"
10. System.out.println(s1.compareTo(s4));//-1 because "l" is 1 times lower than "m"
11. System.out.println(s1.compareTo(s5));//2 because "h" is 2 times greater than "f"
12. }}
```

If you compare string with blank or empty string, it returns length of the string. If second string is empty, result would be positive. If first string is empty, result would be negative.

Example

```
1. public class CompareToExample2{
2. public static void main(String args[]){
3. String s1="hello";
4. String s2="";
5. String s3="me";
6. System.out.println(s1.compareTo(s2));
7. System.out.println(s2.compareTo(s3));
8. }}
```

Java String concat

The **java string concat()** method *combines specified string at the end of this string*. It returns combined string. It is like appending another string.

Example

```
1. public class ConcatExample{
2. public static void main(String args[]){
3. String s1="java string";
4. s1.concat("is immutable");
5. System.out.println(s1);
6. s1=s1.concat(" is immutable so assign it explicitly");
7. System.out.println(s1);
8. }}
```

Example

```
1. public class ConcatExample2 {
2.    public static void main(String[] args) {
3.       String str1 = "Hello";
4.       String str2 = "CMRTC";
5.       String str3 = "CSE";
6.       // Concatenating one string
7.       String str4 = str1.concat(str2);
8.       System.out.println(str4);
9.       // Concatenating multiple strings
10.      String str5 = str1.concat(str2).concat(str3);
11.      System.out.println(str5);
12.   } }
```

Example

```
1. public class ConcatExample3 {
2.    public static void main(String[] args) {
3.       String str1 = "Hello";
4.       String str2 = "CMRTC";
5.       String str3 = "CSE";
6.       // Concatenating Space among strings
7.       String str4 = str1.concat(" ").concat(str2).concat(" ").concat(str3);
8.       System.out.println(str4);
```

```
        // Concatenating Special Chars
        String str5 = str1.concat("!!!");
        System.out.println(str5);
        String str6 = str1.concat("@").concat(str2);
        System.out.println(str6);
    } }
```

Java String contains()

The **java string contains()** method searches the sequence of characters in this string. It returns *true* if sequence of char values are found in this string otherwise returns *false*.

Example

```
class ContainsExample{
public static void main(String args[]){
String name="what do you know about me";
System.out.println(name.contains("do you know"));
System.out.println(name.contains("about"));
System.out.println(name.contains("hello"));
}}
```

Example

```
public class ContainsExample2 {
    public static void main(String[] args) {
        String str = "Hello CMRTC CSE";
        boolean isContains = str.contains("CMRTC");
        System.out.println(isContains);
        // Case Sensitive
        System.out.println(str.contains("cMRTC")); // false
    } }
```

Java String endsWith()

The **java string endsWith()** method checks if this string ends with given suffix. It returns true if this string ends with given suffix else returns false.

Example

```
public class EndsWithExample{
public static void main(String args[]){
String s1="CMRTC CSE";
System.out.println(s1.endsWith("E"));
System.out.println(s1.endsWith("CSE"));
}}
```

Java String equals()

The **java string equals()** method compares the two given strings based on the content of the string. If any character is not matched, it returns false. If all characters are matched, it returns true.

The String equals() method overrides the equals() method of Object class.

Example

```
public class EqualsExample{
public static void main(String args[]){
String s1="CMRTC";
String s2="CMRTC";
String s3="cmrtc";
String s4="python";
System.out.println(s1.equals(s2));//true because content and case is same
System.out.println(s1.equals(s3));//false because case is not same
System.out.println(s1.equals(s4));//false because content is not same
}}
```

Example

```
import java.util.ArrayList;
public class EqualsExample3 {
```

```
    public static void main(String[] args) {
        String str1 = "Mukesh";
        ArrayList<String> list = new ArrayList<>();
        list.add("Ravi");
        list.add("Mukesh");
        list.add("Ramesh");
        list.add("Ajay");
        for (String str : list) {
            if (str.equals(str1)) {
                System.out.println("Mukesh is present");
            } } }}
```

Java String equalsIgnoreCase()

The **String equalsIgnoreCase()** method compares the two given strings on the basis of content of the string irrespective of case of the string. It is like equals() method but doesn't check case. If any character is not matched, it returns false otherwise it returns true.

Example

```
public class EqualsIgnoreCaseExample{
public static void main(String args[]){
String s1="cmrtc";
String s2="cmrtc";
String s3="CMRTC";
String s4="python";
System.out.println(s1.equalsIgnoreCase(s2));//true because content and case both are same
System.out.println(s1.equalsIgnoreCase(s3));//true because case is ignored
System.out.println(s1.equalsIgnoreCase(s4));//false because content is not same
}}
```

Java String format()

The **java string format()** method returns the formatted string by given locale, format and arguments. If you don't specify the locale in String.format() method, it uses default locale by calling *Locale.getDefault()* method.

The format() method of java language is like *sprintf()* function in c language and *printf()* method of java language.

Example

```
public class FormatExample{
public static void main(String args[]){
String name="sonoo";
String sf1=String.format("name is %s",name);
String sf2=String.format("value is %f",32.33434);
String sf3=String.format("value is %32.12f",32.33434);//returns 12 char fractional part filling with 0
System.out.println(sf1);
System.out.println(sf2);
System.out.println(sf3);
```

Example

```
public class FormatExample2 {
    public static void main(String[] args) {
        String str1 = String.format("%d", 101);         // Integer value
        String str2 = String.format("%s", "Amar Singh"); // String value
        String str3 = String.format("%f", 101.00);      // Float value
        String str4 = String.format("%x", 101);         // Hexadecimal value
        String str5 = String.format("%c", 'c');         // Char value
        System.out.println(str1);
        System.out.println(str2);
        System.out.println(str3);
        System.out.println(str4);
        System.out.println(str5);
    } }
```

Example

```
1.  public class FormatExample3 {
2.     public static void main(String[] args) {
3.        String str1 = String.format("%d", 101);
4.        String str2 = String.format("|%10d|", 101);  // Specifying length of integer
5.        String str3 = String.format("|%-10d|", 101); // Left-justifying within the specified width
6.        String str4 = String.format("|% d|", 101);
7.        String str5 = String.format("|%010d|", 101); // Filling with zeroes
8.        System.out.println(str1);
9.        System.out.println(str2);
10.       System.out.println(str3);
11.       System.out.println(str4);
12.       System.out.println(str5);
13.    } }
```

Format Specifier	Data Type	Output
%a	floating point (except *BigDecimal*)	Returns Hex output of floating point number.
%b	Any type	"true" if non-null, "false" if null
%c	character	Unicode character
%d	integer (incl. byte, short, int, long, bigint)	Decimal Integer
%e	floating point	decimal number in scientific notation
%f	floating point	decimal number
%g	floating point	decimal number, possibly in scientific notation depending on the precision and value.
%h	any type	Hex String of value from hashCode() method.
%n	none	Platform-specific line separator.
%o	integer (incl. byte, short, int, long, bigint)	Octal number
%s	any type	String value
%t	Date/Time (incl. long, Calendar, Date and TemporalAccessor)	%t is the prefix for Date/Time conversions. More formatting flags are needed after this. See Date/Time conversion below.
%x	integer (incl. byte, short, int, long, bigint)	Hex string.

Java String getBytes()

The **java string getBytes()** method returns the byte array of the string. In other words, it returns sequence of bytes.

Example

```
1. public class StringGetBytesExample{
2. public static void main(String args[]){
3. String s1="ABCDEFG";
4. byte[] barr=s1.getBytes();
5. for(int i=0;i<barr.length;i++){
6. System.out.println(barr[i]);
7. } }}
```

Example

```
1.  public class StringGetBytesExample2 {
2.     public static void main(String[] args) {
3.        String s1 = "ABCDEFG";
4.        byte[] barr = s1.getBytes();
5.        for(int i=0;i<barr.length;i++){
6.           System.out.println(barr[i]);
7.        }
8.        // Getting string back
9.        String s2 = new String(barr);
10.       System.out.println(s2);
11.    } }
```

Java String getChars()

The **java string getChars()** method copies the content of this string into specified char array. There are 4 arguments passed in getChars() method.

Example

```
1. public class StringGetCharsExample{
2. public static void main(String args[]){
3.  String str = new String("hello CMRTC CSE how r u");
4.      char[] ch = new char[10];
5.      try{
6.        str.getChars(6, 16, ch, 0);
7.        System.out.println(ch);
8.      }catch(Exception ex){System.out.println(ex);}
9. }}
```

Java String indexOf()

The **java string indexOf()** method returns index of given character value or substring. If it is not found, it returns -1. The index counter starts from zero.

No.	Method	Description
1	int indexOf(int ch)	returns index position for the given char value
2	int indexOf(int ch, int fromIndex)	returns index position for the given char value and from index
3	int indexOf(String substring)	returns index position for the given substring
4	int indexOf(String substring, int fromIndex)	returns index position for the given substring and from index

Example

```
1. public class IndexOfExample{
2. public static void main(String args[]){
3. String s1="this is index of example";
4. //passing substring
5. int index1=s1.indexOf("is");//returns the index of is substring
6. int index2=s1.indexOf("index");//returns the index of index substring
7. System.out.println(index1+"  "+index2);//2 8
8.  //passing substring with from index
9. int index3=s1.indexOf("is",4);//returns the index of is substring after 4th index
10. System.out.println(index3);//5 i.e. the index of another is
11. //passing char value
12. int index4=s1.indexOf('s');//returns the index of s char value
13. System.out.println(index4);//3
14. }}
```

Example

```
1. public class IndexOfExample3 {
2.    public static void main(String[] args) {
3.       String s1 = "This is indexOf method";
4.       // Passing substring and index
5.       int index = s1.indexOf("method", 10); //Returns the index of this substring
6.       System.out.println("index of substring "+index);
7.       index = s1.indexOf("method", 20); // It returns -1 if substring does not found
8.       System.out.println("index of substring "+index);
9.    } }
```

Java String intern()

The **java string intern()** method returns the interned string. It returns the canonical representation of string.

It can be used to return string from memory, if it is created by new keyword. It creates exact copy of heap string object in string constant pool.

Example

```
1. public class InternExample{
2. public static void main(String args[]){
3. String s1=new String("hello");
4. String s2="hello";
5. String s3=s1.intern();//returns string from pool, now it will be same as s2
6. System.out.println(s1==s2);//false because reference variables are pointing to different instance
7. System.out.println(s2==s3);//true because reference variables are pointing to same instance
8. }}
```

Java String isEmpty()

The **java string isEmpty()** method checks if this string is empty or not. It returns *true*, if length of string is 0 otherwise *false*. In other words, true is returned if string is empty otherwise it returns false.

Example

```
1. public class IsEmptyExample{
2. public static void main(String args[]){
3. String s1="";
4. String s2="CMRTC CSE";
5.  System.out.println(s1.isEmpty());
6. System.out.println(s2.isEmpty());
7. }}
```

Java String join()

The **java string join()** method returns a string joined with given delimiter. In string join method, delimiter is copied for each elements.

In case of null element, "null" is added.

Example

```
1. public class StringJoinExample{
2. public static void main(String args[]){
3. String joinString1=String.join("-","welcome","to","CMRTC CSE");
4. System.out.println(joinString1);
5. }}
```

Example

```
1. public class StringJoinExample2 {
2.    public static void main(String[] args) {
3.       String date = String.join("/","25","06","2018");
4.       System.out.print(date);
5.       String time = String.join(":", "12","10","10");
6.       System.out.println(" "+time);
7.    } }
```

Java String lastIndexOf()

The **java string lastIndexOf()** method returns last index of the given character value or substring. If it is not found, it returns -1. The index counter starts from zero.
There are 4 types of lastIndexOf method in java. The signature of lastIndexOf methods are given below:

No.	Method	Description
1	int lastIndexOf(int ch)	returns last index position for the given char value
2	int lastIndexOf(int ch, int fromIndex)	returns last index position for the given char value and from index
3	int lastIndexOf(String substring)	returns last index position for the given substring
4	int lastIndexOf(String substring, int fromIndex)	returns last index position for the given substring and from index

Example

```
1. public class LastIndexOfExample{
2. public static void main(String args[]){
3. String s1="this is index of example";//there are 2 's' characters in this sentence
4. int index1=s1.lastIndexOf('s');//returns last index of 's' char value
5. System.out.println(index1);//6
6. }}
```

Example

```
1. public class LastIndexOfExample2 {
2.    public static void main(String[] args) {
3.       String str = "This is index of example";
4.       int index = str.lastIndexOf('s',5);
5.       System.out.println(index);
6.    } }
```

Example

```
1. public class LastIndexOfExample3 {
2.    public static void main(String[] args) {
3.       String str = "This is last index of example";
4.       int index = str.lastIndexOf("of");
5.       System.out.println(index);
6.    } }
```

Example

```
1. public class LastIndexOfExample4 {
2.    public static void main(String[] args) {
```

```
        String str = "This is last index of example";
        int index = str.lastIndexOf("of", 25);
        System.out.println(index);
        index = str.lastIndexOf("of", 10);
        System.out.println(index); // -1, if not found
    } }
```

Java String length()

The **java string length()** method length of the string. It returns count of total number of characters. The length of java string is same as the unicode code units of the string.

Example

```
public class LengthExample{
public static void main(String args[]){
String s1="CMRTC CSE";
String s2="JAVA";
System.out.println("string length is: "+s1.length());//9 is the length of javatpoint string
System.out.println("string length is: "+s2.length());//4 is the length of python string
}}
```

Example

```
public class LengthExample2 {
    public static void main(String[] args) {
        String str = "Javatpoint";
        if(str.length()>0) {
            System.out.println("String is not empty and length is: "+str.length());
        }
        str = "";
        if(str.length()==0) {
            System.out.println("String is empty now: "+str.length());
        } } }
```

Java String replace()

The **java string replace()** method returns a string replacing all the old char or CharSequence to new char or CharSequence.

Example

```
public class ReplaceExample1{
public static void main(String args[]){
String s1="CMRTC is a very good Engineering College";
String replaceString=s1.replace('a','e');//replaces all occurrences of 'a' to 'e'
System.out.println(replaceString);
}}
```

Example

```
public class ReplaceExample2{
public static void main(String args[]){
String s1="my name is khan my name is java";
String replaceString=s1.replace("is","was");//replaces all occurrences of "is" to "was"
System.out.println(replaceString);
}}
```

Example

```
public class ReplaceExample3 {
    public static void main(String[] args) {
        String str = "oooooo-hhhh-oooooo";
        String rs = str.replace("h","s"); // Replace 'h' with 's'
        System.out.println(rs);
        rs = rs.replace("s","h"); // Replace 's' with 'h'
        System.out.println(rs);
    } }
```

Java String replaceAll()

The **java string replaceAll()** method returns a string replacing all the sequence of characters matching regex and replacement string.

Example

```
1. public class ReplaceAllExample1{
2. public static void main(String args[]){
3. String s1="CMRTC is a very good College";
4. String replaceString=s1.replaceAll("e","a");//replaces all occurrences of "e" to "a"
5. System.out.println(replaceString);
6. }}
```

Example

```
1. public class ReplaceAllExample2{
2. public static void main(String args[]){
3. String s1="My name is Khan. My name is Bob. My name is Sonoo.";
4. String replaceString=s1.replaceAll("is","was");//replaces all occurrences of "is" to "was"
5. System.out.println(replaceString);
6. }}
```

Example

```
1. public class ReplaceAllExample3{
2. public static void main(String args[]){
3. String s1="My name is Khan. My name is Bob. My name is Sonoo.";
4. String replaceString=s1.replaceAll("\\s","");
5. System.out.println(replaceString);
6. }}
```

Java String split()

The **java string split()** method splits this string against given regular expression and returns a char array.

Example

```
1. public class SplitExample{
2. public static void main(String args[]){
3. String s1="java string split method by JAVA";
4. String[] words=s1.split("\\s");//splits the string based on whitespace
5. //using java foreach loop to print elements of string array
6. for(String w:words){
7. System.out.println(w);
8. }  }}
```

Example

```
1. public class SplitExample2{
2. public static void main(String args[]){
3. String s1="welcome to split world";
4. System.out.println("returning words:");
5. for(String w:s1.split("\\s",0)){
6. System.out.println(w);
7. }
8. System.out.println("returning words:");
9. for(String w:s1.split("\\s",1)){
10. System.out.println(w);
11. }
12. System.out.println("returning words:");
13. for(String w:s1.split("\\s",2)){
14. System.out.println(w);
15. }  }}
```

Java String startsWith()

The **java string startsWith()** method checks if this string starts with given prefix. It returns true if this string starts with given prefix else returns false.

Example

```
1. public class StartsWithExample{
2. public static void main(String args[]){
3. String s1="java string split method by javatpoint";
4. System.out.println(s1.startsWith("ja"));
5. System.out.println(s1.startsWith("java string"));
6. }}
```

Example

```
1. public class StartsWithExample2 {
2.    public static void main(String[] args) {
3.       String str = "Java";
4.       System.out.println(str.startsWith("J")); // True
5.       System.out.println(str.startsWith("a")); // False
6.       System.out.println(str.startsWith("a",1)); // True
7.    } }
```

Java String substring()

The **java string substring()** method returns a part of the string.

We pass begin index and end index number position in the java substring method where start index is inclusive and end index is exclusive. In other words, start index starts from 0 whereas end index starts from 1.

Example

```
1. public class SubstringExample{
2. public static void main(String args[]){
3. String s1="CMRTC CSE";
4. System.out.println(s1.substring(2,4));//returns RT
5. System.out.println(s1.substring(2));//returns RTC CSE
6. }}
```

Example

```
1. public class SubstringExample2 {
2.    public static void main(String[] args) {
3.       String s1="CMRTC CSE";
4.       String substr = s1.substring(0); // Starts with 0 and goes to end
5.       System.out.println(substr);
6.       String substr2 = s1.substring(5,8); // Starts from 5 and goes to 8
7.       System.out.println(substr2);
8.       String substr3 = s1.substring(5,15); // Returns Exception
9.    } }
```

Java String toCharArray()

The **java string toCharArray()** method converts this string into character array. It returns a newly created character array, its length is similar to this string and its contents are initialized with the characters of this string.

Example

```
1. public class StringToCharArrayExample{
2. public static void main(String args[]){
3. String s1="hello";
4. char[] ch=s1.toCharArray();
5. for(int i=0;i<ch.length;i++){
6. System.out.print(ch[i]);
7. } }}
```

Example

```
1. public class StringToCharArrayExample2 {
2.    public static void main(String[] args) {
3.       String s1 = "Welcome to Javatpoint";
4.       char[] ch = s1.toCharArray();
5.       int len = ch.length;
6.       System.out.println("Char Array length: " + len);
7.       System.out.println("Char Array elements: ");
```

```
8.        for (int i = 0; i < len; i++) {
9.            System.out.println(ch[i]);
10.       } } }
```

Java String toLowerCase()

The **java string toLowerCase()** method returns the string in lowercase letter. In other words, it converts all characters of the string into lower case letter.

The toLowerCase() method works same as toLowerCase(Locale.getDefault()) method. It internally uses the default locale

Example

```
1. public class StringLowerExample{
2. public static void main(String args[]){
3. String s1="JAVA HELLO stRIng";
4. String s1lower=s1.toLowerCase();
5. System.out.println(s1lower);
6. }}
```

Example

```
1. import java.util.Locale;
2. public class StringLowerExample2 {
3.     public static void main(String[] args) {
4.         String s = "JAVATPOINT HELLO stRIng";
5.         String eng = s.toLowerCase(Locale.ENGLISH);
6.         System.out.println(eng);
7.         String turkish = s.toLowerCase(Locale.forLanguageTag("tr")); // It shows i without dot
8.         System.out.println(turkish);
9.     } }
```

Java String toUpperCase()

The **java string toUpperCase()** method returns the string in uppercase letter. In other words, it converts all characters of the string into upper case letter.

The toUpperCase() method works same as toUpperCase(Locale.getDefault()) method. It internally uses the default locale.

Example

```
1. public class StringUpperExample{
2. public static void main(String args[]){
3. String s1="hello string";
4. String s1upper=s1.toUpperCase();
5. System.out.println(s1upper);
6. }}
```

Example

```
1. import java.util.Locale;
2. public class StringUpperExample2 {
3.     public static void main(String[] args) {
4.         String s = "hello string";
5.         String turkish = s.toUpperCase(Locale.forLanguageTag("tr"));
6.         String english = s.toUpperCase(Locale.forLanguageTag("en"));
7.         System.out.println(turkish);//will print I with dot on upper side
8.         System.out.println(english);
9.     } }
```

Java String trim()

The **java string trim()** method eliminates leading and trailing spaces. The unicode value of space character is '\u0020'. The trim() method in java string checks this unicode value before and after the string, if it exists then removes the spaces and returns the omitted string.

Example

```
1. public class StringTrimExample{
2. public static void main(String args[]){
3. String s1="  hello string   ";
```

```
4. System.out.println(s1+"javatpoint");//without trim()
5. System.out.println(s1.trim()+"javatpoint");//with trim()
6. }}
```

Java String valueOf()

The **java string valueOf()** method converts different types of values into string. By the help of string valueOf() method, you can convert int to string, long to string, boolean to string, character to string, float to string, double to string, object to string and char array to string.

Example

```
1. public class StringValueOfExample5 {
2.     public static void main(String[] args) {
3.         boolean b1=true;
4.         byte b2=11;
5.         short sh = 12;
6.         int i = 13;
7.         long l = 14L;
8.         float f = 15.5f;
9.         double d = 16.5d;
10.        char chr[]={'j','a','v','a'};
11.        StringValueOfExample5 obj=new StringValueOfExample5();
12.        String s1 = String.valueOf(b1);
13.        String s2 = String.valueOf(b2);
14.        String s3 = String.valueOf(sh);
15.        String s4 = String.valueOf(i);
16.        String s5 = String.valueOf(l);
17.        String s6 = String.valueOf(f);
18.        String s7 = String.valueOf(d);
19.        String s8 = String.valueOf(chr);
20.        String s9 = String.valueOf(obj);
21.        System.out.println(s1);
22.        System.out.println(s2);
23.        System.out.println(s3);
24.        System.out.println(s4);
25.        System.out.println(s5);
26.        System.out.println(s6);
27.        System.out.println(s7);
28.        System.out.println(s8);
29.        System.out.println(s9);
30.    }
31. }
```

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns a formatted string.
4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale.
5	String substring(int beginIndex)	returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string.
9	static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)	returns a joined string.
10	boolean equals(Object another)	checks the equality of string with the given object.
11	boolean isEmpty()	checks if string is empty.
12	String concat(String str)	concatenates the specified string.
13	String replace(char old, char new)	replaces all occurrences of the specified char value.
14	String replace(CharSequence old, CharSequence new)	replaces all occurrences of the specified CharSequence.
15	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.

16	String[] split(String regex)	returns a split string matching regex.
17	String[] split(String regex, int limit)	returns a split string matching regex and limit.
18	String intern()	returns an interned string.
19	int indexOf(int ch)	returns the specified char value index.
20	int indexOf(int ch, int fromIndex)	returns the specified char value index starting with given index.
21	int indexOf(String substring)	returns the specified substring index.
22	int indexOf(String substring, int fromIndex)	returns the specified substring index starting with given index.
23	String toLowerCase()	returns a string in lowercase.
24	String toLowerCase(Locale l)	returns a string in lowercase using specified locale.
25	String toUpperCase()	returns a string in uppercase.
26	String toUpperCase(Locale l)	returns a string in uppercase using specified locale.
27	String trim()	removes beginning and ending spaces of this string.
28	static String valueOf(int value)	converts given type into string. It is an overloaded method.

Inheritance

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

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

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

Advantages

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

Terms in Inheritance

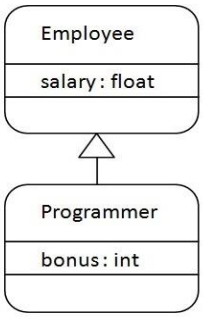
- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **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. You can use the same fields and methods already defined in the previous class.

Syntax

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

The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.

In the terminology of Java, a class which is inherited is called a parent or superclass, and the new class is called child or subclass.



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**.

It means that Programmer is a type of Employee.

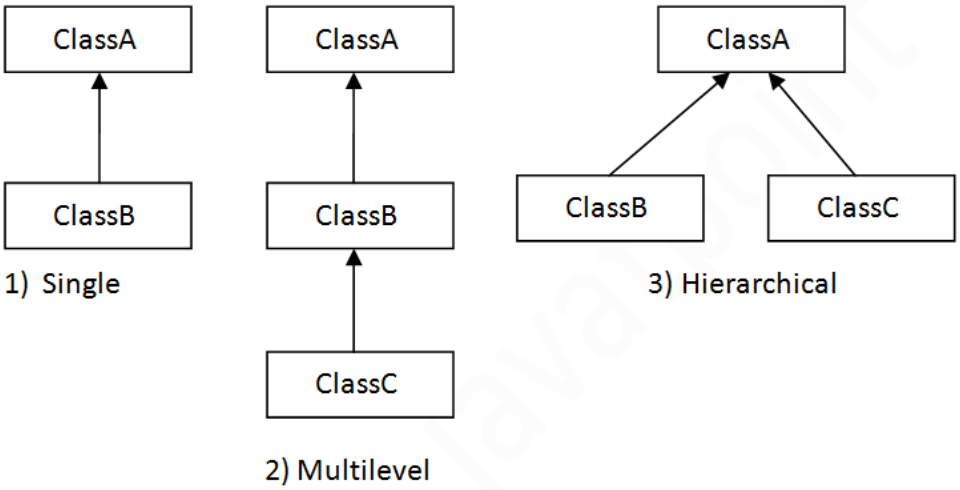
```
class Employee{
 float salary=40000;
}
```

```
4. class Programmer extends Employee{
5.  int bonus=10000;
6.  public static void main(String args[]){
7.    Programmer p=new Programmer();
8.    System.out.println("Programmer salary is:"+p.salary);
9.    System.out.println("Bonus of Programmer is:"+p.bonus);
10. }  }
```

Types of inheritance in java

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

In java programming, multiple and hybrid inheritance is supported through interface only.



Single Inheritance Example

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class single{
8. public static void main(String args[]){
9. Dog d=new Dog();
10. d.bark();
11. d.eat();
12. }}
```

Multilevel Inheritance Example

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class BabyDog extends Dog{
```

```
8. void weep(){System.out.println("weeping...");}
9. }
10. class multilevel{
11. public static void main(String args[]){
12. BabyDog d=new BabyDog();
13. d.weep();
14. d.bark();
15. d.eat();
16. }}
```

Hierarchical Inheritance Example

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void bark(){System.out.println("barking...");}
6. }
7. class Cat extends Animal{
8. void meow(){System.out.println("meowing...");}
9. }
10. class hierarchical{
11. public static void main(String args[]){
12. Cat c=new Cat();
13. c.meow();
14. c.eat();
15. //c.bark();//C.T.Error
16. }}
```

multiple inheritance is not supported in java

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
1. class A{
2. void msg(){System.out.println("Hello");}
3. }
4. class B{
5. void msg(){System.out.println("Welcome");}
6. }
7. class C extends A,B{//suppose if it were
8.
9.  public static void main(String args[]){
10.   C obj=new C();
11.   obj.msg();//Now which msg() method would be invoked?
12. } }
```

Access Modifiers in Java

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1) Private

The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

Example

```
1. class A{
2. private int data=40;
3. private void msg(){System.out.println("Hello java");}
4. }
5. public class Simple{
6.  public static void main(String args[]){
7.    A obj=new A();
8.    System.out.println(obj.data);//Compile Time Error
9.    obj.msg();//Compile Time Error
10.   }  }
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

```
1. class A{
2. private A(){}//private constructor
3. void msg(){System.out.println("Hello java");}
4. }
5. public class Simple{
6.  public static void main(String args[]){
7.    A obj=new A();//Compile Time Error
8.  }  }
```

Note: A class cannot be private or protected except nested class.

2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

```
1. //save by A.java
2. package pack;
3. class A{
```

```
4.   void msg(){System.out.println("Hello");}
5.  }
```

```
1.  //save by B.java
2.  package mypack;
3.  import pack.*;
4.  class B{
5.    public static void main(String args[]){
6.     A obj = new A();//Compile Time Error
7.     obj.msg();//Compile Time Error
8.    }  }
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifer.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```
1.  //save by A.java
2.  package pack;
3.  public class A{
4.  protected void msg(){System.out.println("Hello");}
5.  }
```

```
1.  //save by B.java
2.  package mypack;
3.  import pack.*;
4.  class B extends A{
5.    public static void main(String args[]){
6.     B obj = new B();
7.     obj.msg();
8.    }  }
```

4) Public

The **public access modifier** is accessible everywhere. It has the widest scope among all other modifiers.

Example of public access modifier

```
1.  //save by A.java
2.  package pack;
```

```
3. public class A{
4. public void msg(){System.out.println("Hello");}
5. }
```

```
1. //save by B.java
2. package mypack;
3. import pack.*;
4. class B{
5.   public static void main(String args[]){
6.    A obj = new A();
7.    obj.msg();
8.   }
9. }
```

Java Access Modifiers with Method Overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

```
1. class A{
2. protected void msg(){System.out.println("Hello java");}
3. }
4. public class Simple extends A{
5. void msg(){System.out.println("Hello java");}//C.T.Error
6.  public static void main(String args[]){
7.    Simple obj=new Simple();
8.    obj.msg();
9.    }  }
```

Note: The default modifier is more restrictive than protected. That is why, there is a compile-time error.

access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Constructors

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

It is a special type of method which is used to initialize the object. Every time an object is created using the new() keyword, at least one constructor is called.

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

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

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

Rules for creating Java constructor

There are 3 rules defined for the constructor.

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

Note: We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

Types of Java constructors

There are two types of constructors in Java:

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

Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.
Syntax:
<class_name>(){}

Example:

```
1. //Java Program to create and call a default constructor
2. class Bike1{
3. //creating a default constructor
4. Bike1(){System.out.println("Bike is created");}
5. //main method
6. public static void main(String args[]){
7. //calling a default constructor
8. Bike1 b=new Bike1();
9. } }
```

Rule: If there is no constructor in a class, compiler automatically creates a default constructor.

The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.

Example:

```
1. //Let us see another example of default constructor
2. //which displays the default values
3. class Student3{
4. int id;
5. String name;
6. //method to display the value of id and name
7. void display(){System.out.println(id+" "+name);}
8. public static void main(String args[]){
9. //creating objects
```

```
Student3 s1=new Student3();
Student3 s2=new Student3();
//displaying values of the object
s1.display();
s2.display();
} }
```

We are not created any constructor so compiler provides a default constructor. Here 0 and null values are provided by default constructor.

Parameterized Constructor

A constructor which has a specific number of parameters is called a parameterized constructor. The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example:

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}
    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   } }
```

Constructor Overloading

In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.

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. They are differentiated by the compiler by the number of parameters in the list and their types.

Example:

```
//Java program to overload constructors
class Student5{
    int id;
    String name;
    int age;
    //creating two arg constructor
    Student5(int i,String n){
    id = i;
    name = n;
    }
    //creating three arg constructor
    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,"Karan");
    Student5 s2 = new Student5(222,"Aryan",25);
    s1.display();
    s2.display();
   } }
```

There are many differences between constructors and methods. They are given below.

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 same as the class name.	The method name may or may not be same as the class name.

Java Copy Constructor

There is no copy constructor in Java. However, we can copy the values from one object to another like copy constructor in C++.

There are many ways to copy the values of one object into another in Java. They are:

- By constructor
- By assigning the values of one object into another
- By clone() method of Object class

In this example, we are going to copy the values of one object into another using Java constructor.

```
//Java program to initialize the values from one object to another object.
class Student6{
    int id;
    String name;
    //constructor to initialize integer and string
    Student6(int i,String n){
    id = i;
    name = n;
    }
    //constructor to initialize another object
    Student6(Student6 s){
    id = s.id;
    name =s.name;
    }
    void display(){System.out.println(id+" "+name);}
    public static void main(String args[]){
    Student6 s1 = new Student6(111,"Karan");
    Student6 s2 = new Student6(s1);
```

```
    s1.display();
    s2.display();
  } }
```

Copying values without constructor

We can copy the values of one object into another by assigning the objects values to another object. In this case, there is no need to create the constructor.

Example:

```
class Student7{
    int id;
    String name;
    Student7(int i,String n){
    id = i;
    name = n;
    }
    Student7(){}
    void display(){System.out.println(id+" "+name);}
    public static void main(String args[]){
    Student7 s1 = new Student7(111,"Karan");
    Student7 s2 = new Student7();
    s2.id=s1.id;
    s2.name=s1.name;
    s1.display();
    s2.display();
  } }
```

Super Keyword in Java

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

Example:

```
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();
}}
```

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

Example:

```
1. class Animal{
2. void eat(){System.out.println("eating...");}
3. }
4. class Dog extends Animal{
5. void eat(){System.out.println("eating bread...");}
6. void bark(){System.out.println("barking...");}
7. void work(){
8. super.eat();
9. bark();
10. } }
11. class TestSuper2{
12. public static void main(String args[]){
13. Dog d=new Dog();
14. d.work();
15. }}
```

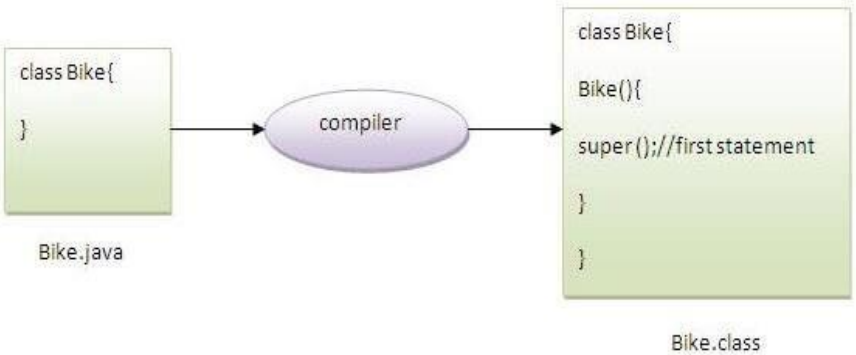
3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor.

Example:

```
1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
6. super();
7. System.out.println("dog is created");
8. } }
9. class TestSuper3{
10. public static void main(String args[]){
11. Dog d=new Dog();
12. }}
```

Note: super() is added in each class constructor automatically by compiler if there is no super() or this().



default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.

Example:

```
1. class Animal{
2. Animal(){System.out.println("animal is created");}
3. }
4. class Dog extends Animal{
5. Dog(){
6. System.out.println("dog is created");
7. } }
8. class TestSuper4{
9. public static void main(String args[]){
10. Dog d=new Dog();
11. }}
```

Example:

```
1. class Person{
2. int id;
3. String name;
4. Person(int id,String name){
5. this.id=id;
6. this.name=name;
7. } }
8. class Emp extends Person{
9. float salary;
10. Emp(int id,String name,float salary){
11. super(id,name);//reusing parent constructor
12. this.salary=salary;
13. }
14. void display(){System.out.println(id+" "+name+" "+salary);}
15. }
16. class TestSuper5{
17. public static void main(String[] args){
18. Emp e1=new Emp(1,"ankit",45000f);
19. e1.display();
20. }}
```

Final Keyword In Java

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

1. variable
2. method
3. class

The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only.

1) Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Example:

```
1. class Bike9{
2.  final int speedlimit=90;//final variable
3.  void run(){
4.   speedlimit=400;
5.  }
6.  public static void main(String args[]){
7.  Bike9 obj=new  Bike9();
8.  obj.run();
9.  } }
```

2) Java final method

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

Example:

```
1. class Bike{
2.   final void run(){System.out.println("running");}
3. }
4. class Honda extends Bike{
5.    void run(){System.out.println("running safely with 100kmph");}
6.    public static void main(String args[]){
7.    Honda honda= new Honda();
8.    honda.run();
9.    } }
```

3) Java final class

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

Example:

```
1. final class Bike{}
2. class Honda1 extends Bike{
3.   void run(){System.out.println("running safely with 100kmph");}
4.   public static void main(String args[]){
5.   Honda1 honda= new Honda1();
6.   honda.run();
7.   } }
```

final method is inherited but you cannot override it. For Example:

```
1. class Bike{
2.   final void run(){System.out.println("running...");}
3. }
4. class Honda2 extends Bike{
5.    public static void main(String args[]){
6.     new Honda2().run();
7.    }
8. }
```

blank or uninitialized final variable

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. It can be initialized only in constructor.

For example PAN CARD number of an employee.

Example:

```
1. class Bike10{
2.   final int speedlimit;//blank final variable
3.   Bike10(){
4.   speedlimit=70;
5.   System.out.println(speedlimit);
6.   }
7.   public static void main(String args[]){
8.     new Bike10();
9.  } }
```

static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.

Example:

```
1. class A{
2.   static final int data;//static blank final variable
3.   static{ data=50;}
4.   public static void main(String args[]){
5.     System.out.println(A.data);
6.  }  }
```

final parameter

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

Example:

```
1. class Bike11{
2.   int cube(final int n){
3.    n=n+2;//can't be changed as n is final
4.    n*n*n;
5.    return n;
6.   }
7.   public static void main(String args[]){
8.     Bike11 b=new Bike11();
9.     b.cube(5);
10. }  }
```

Note: Constructor can't be declared as final because constructor is never inherited.

Method Overloading in Java

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

If we have to perform only one operation, having same name of the methods increases the readability of the program.

Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

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

Note: ***In java, Method Overloading is not possible by changing the return type of the method only.***

1) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static int add(int a,int b,int c){return a+b+c;}
4. }
5. class TestOverloading1{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(11,11,11));
9. }}
```

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
1. class Adder{
2. static int add(int a, int b){return a+b;}
3. static double add(double a, double b){return a+b;}
4. }
5. class TestOverloading2{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));
8. System.out.println(Adder.add(12.3,12.6));
9. }}
```

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

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

```
1. class Adder{
2. static int add(int a,int b){return a+b;}
3. static double add(int a,int b){return a+b;}
4. }
5. class TestOverloading3{
6. public static void main(String[] args){
7. System.out.println(Adder.add(11,11));//ambiguity
8. }}
```

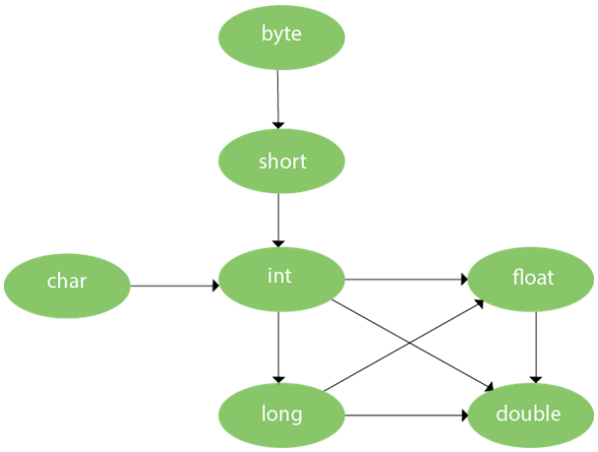
Note: Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.

by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

```
1. class TestOverloading4{
2. public static void main(String[] args){System.out.println("main with String[]");}
3. public static void main(String args){System.out.println("main with String");}
4. public static void main(){System.out.println("main without args");}
5. }
```

Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int,long,float or double. The char datatype can be promoted to int,long,float or double and so on.

Example of Method Overloading with TypePromotion

```
1. class OverloadingCalculation1{
2.   void sum(int a,long b){System.out.println(a+b);}
3.   void sum(int a,int b,int c){System.out.println(a+b+c);}
4.   public static void main(String args[]){
5.   OverloadingCalculation1 obj=new OverloadingCalculation1();
6.   obj.sum(20,20);//now second int literal will be promoted to long
7.   obj.sum(20,20,20);
8.   } }
```

Example of Method Overloading with Type Promotion if matching found

If there are matching type arguments in the method, type promotion is not performed.

```
1.   class OverloadingCalculation2{
2.   void sum(int a,int b){System.out.println("int arg method invoked");}
3.   void sum(long a,long b){System.out.println("long arg method invoked");}
4.   public static void main(String args[]){
5.   OverloadingCalculation2 obj=new OverloadingCalculation2();
6.   obj.sum(20,20);//now int arg sum() method gets invoked
7.   } }
```

Example of Method Overloading with Type Promotion in case of ambiguity

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
1. class OverloadingCalculation3{
2.   void sum(int a,long b){System.out.println("a method invoked");}
3.   void sum(long a,int b){System.out.println("b method invoked");}
4.   public static void main(String args[]){
5.   OverloadingCalculation3 obj=new OverloadingCalculation3();
6.   obj.sum(20,20);//now ambiguity
7.   } }
```

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java 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).

Example:

```
1. //Java Program to illustrate the use of Java Method Overriding
2. //Creating a parent class.
3. class Vehicle{
4.   //defining a method
5.   void run(){System.out.println("Vehicle is running");}
6. }
7. //Creating a child class
8. class Bike2 extends Vehicle{
9.   //defining the same method as in the parent class
10.  void run(){System.out.println("Bike is running safely");}
11.  public static void main(String args[]){
12.  Bike2 obj = new Bike2();//creating object
13.  obj.run();//calling method
14.  } }
```

Note: ***Java method overriding is mostly used in Runtime Polymorphism***

a static method cannot be overridden. It can be proved by runtime polymorphism. the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

We can not override java method because the main is a static method.

Polymorphism in Java

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Real life example of polymorphism: A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, So it means many forms.

In Java polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism

Compile time polymorphism: It is also known as static polymorphism. This type of polymorphism is achieved by **function overloading** or **operator overloading**.

Method Overloading: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

Example: By using different types of arguments

```
// Java program for Method overloading

class MultiplyFun {

	// Method with 2 parameter
	static int Multiply(int a, int b)
	{
		return a * b;
	}

	// Method with the same name but 2 double parameter
	static double Multiply(double a, double b)
	{
		return a * b;
	}
}

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

		System.out.println(MultiplyFun.Multiply(2, 4));

		System.out.println(MultiplyFun.Multiply(5.5, 6.3));
	}
}
```

Output:

```
8

34.65
```

Example: By using different numbers of arguments

```
// Java program for Method overloading

class MultiplyFun {

	// Method with 2 parameter
	static int Multiply(int a, int b)
	{
		return a * b;
	}

	// Method with the same name but 3 parameter
	static int Multiply(int a, int b, int c)
	{
		return a * b * c;
	}
}
```

```
class Main {
	public static void main(String[] args)
	{
		System.out.println(MultiplyFun.Multiply(2, 4));

		System.out.println(MultiplyFun.Multiply(2, 7, 3));
	}
}
```

Output:

```
8

42
```

Operator Overloading: Java also provide option to overload operators.
For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator ‘+’ when placed between integer operands, adds them and when placed between string operands, concatenates them.

In java, Only “+” operator can be overloaded:

- To add integers
- To concatenate strings

Example:

```
// Java program for Operator overloading

class OperatorOVERDDN {

	void operator(String str1, String str2)
	{
		String s = str1 + str2;
		System.out.println("Concatinated String - "
						+ s);
	}

	void operator(int a, int b)
	{
		int c = a + b;
		System.out.println("Sum = " + c);
	}
}

class Main {
	public static void main(String[] args)
	{
		OperatorOVERDDN obj = new OperatorOVERDDN();
		obj.operator(2, 3);
		obj.operator("joe", "now");
	}
}
```

Output:

```
Sum = 5

Concatinated String - joenow
```

Runtime polymorphism: It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.

Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

Example:

```
// Java program for Method overridding

class Parent {

	void Print()
	{
		System.out.println("parent class");
	}
}

class subclass1 extends Parent {

	void Print()
	{
		System.out.println("subclass1");
	}
}

class subclass2 extends Parent {

	void Print()
	{
		System.out.println("subclass2");
	}
}

class TestPolymorphism3 {
	public static void main(String[] args)
	{

		Parent a;

		a = new subclass1();
		a.Print();

		a = new subclass2();
		a.Print();
	}
}
```

Output:

```
subclass1

subclass2
```

Abstract class in Java

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only essential things to the user and hides the internal details, for example, sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.
There are two ways to achieve abstraction in java

1. Abstract class
2. Interface

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot 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.

Example

```
abstract class A{}
```

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

```
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();
} }
```

Example

```
abstract class Shape{
abstract void draw();
}
//In real scenario, implementation is provided by others i.e. unknown by end user
class Rectangle extends Shape{
void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape{
void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
public static void main(String args[]){
Shape s=new Circle1();//In a real scenario, object is provided through method, e.g., getShape() method
s.draw();
} }
```

Example

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
```

```
8.  int getRateOfInterest(){return 8;}
9.  }
10. class TestBank{
11. public static void main(String args[]){
12. Bank b;
13. b=new SBI();
14. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
15. b=new PNB();
16. System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
17. }}
```

An abstract class can have a data member, abstract method, method body (non-abstract method), constructor, and even main() method.

Example

```
1.  //Example of an abstract class that has abstract and non-abstract methods
2.   abstract class Bike{
3.     Bike(){System.out.println("bike is created");}
4.     abstract void run();
5.     void changeGear(){System.out.println("gear changed");}
6.   }
7.  //Creating a Child class which inherits Abstract class
8.   class Honda extends Bike{
9.   void run(){System.out.println("running safely..");}
10.  }
11. //Creating a Test class which calls abstract and non-abstract methods
12.  class TestAbstraction2{
13.  public static void main(String args[]){
14.   Bike obj = new Honda();
15.   obj.run();
16.   obj.changeGear();
17.  }  }
```

Note: ***If there is an abstract method in a class, that class must be abstract.***

Note: ***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.***

Object class in Java

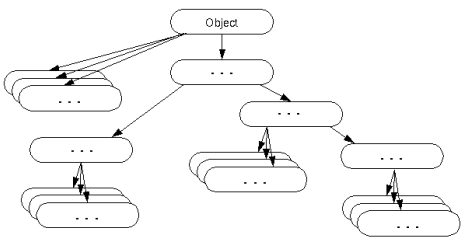
The **Object class** is the parent class of all the classes in java by default. In other words, it is the topmost class of java.

The Object class is beneficial if you want to refer any object whose type you don't know. Notice that parent class reference variable can refer the child class object, know as upcasting.

Let's take an example, there is getObject() method that returns an object but it can be of any type like Employee,Student etc, we can use Object class reference to refer that object.
For example:

```
Object obj=getObject();//we don't know what object will be returned from this method
```

The Object class provides some common behaviors to all the objects such as object can be compared, object can be cloned, object can be notified etc.



Methods of Object class

The Object class provides many methods. They are as follows:

Method	Description
public final Class getClass()	returns the Class class object of this object. The Class class can further be used to get the metadata of this class.
public int hashCode()	returns the hashcode number for this object.
public boolean equals(Object obj)	compares the given object to this object.
protected Object clone() throws CloneNotSupportedException	creates and returns the exact copy (clone) of this object.
public String toString()	returns the string representation of this object.
public final void notify()	wakes up single thread, waiting on this object's monitor.
public final void notifyAll()	wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout)throws InterruptedException	causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos)throws InterruptedException	causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait()throws InterruptedException	causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize()throws Throwable	is invoked by the garbage collector before object is being garbage collected.

Forms of Inheritance

All objects eventually inherit from `Object`, which provides useful methods such as `equals` and `toString`.

In general we want to satisfy *substitutability:* if B is a subclass of A, anywhere we expect an instance of A we can use an instance of B.

Inheritance gets used for a number of purposes in typical object-oriented programming:

specialization -- the subclass is a special case of the parent class (e.g. Frame and CannonWorld)

A good example is the Java hierarchy of Graphical components in the AWT:

- Component
  - Label
  - Button
  - TextComponent
    - TextArea
    - TextField
  - CheckBox
  - ScrollBar

Each child class overrides a method inherited from the parent in order to specialize the class in some way.

specification -- the superclass just specifies which methods should be available but doesn't give code. This is supported in java by interfaces and abstract methods.

If the parent class is abstract, we often say that it is providing a specification for the child class, and therefore it is specification inheritance (a variety of specialization inheritance).

Example: Java Event Listeners

ActionListener, MouseListener, and so on specify behavior, but must be subclassed.

construction -- the superclass is just used to provide behavior, but instances of the subclass don't really act like the superclass. Violates substitutability.

Exmample: defining Stack as a subclass of Vector. This is not clean -- better to define Stack as having a field that holds a vector.

If the parent class is used as a source for behavior, but the child class has no *is-a* relationship to the parent, then we say the child class is using inheritance for construction.

An example might be subclassing the idea of a **Set** from an existing **List** class.

Generally not a good idea, since it can break the principle of substituability, but nevertheless sometimes found in practice. (More often in dynamically typed languages, such as Smalltalk).

extension -- subclass adds new methods, and perhaps redefines inherited ones as well.

If a child class generalizes or extends the parent class by providing more functionality, but does not override any method, we call it inheritance for generalization.

The child class doesn't change anything inherited from the parent, it simply adds new features.

An example is Java Properties inheriting form Hashtable.

limitation -- the subclass restricts the inherited behavior. Violates substitutability. Example: defining Queue as a subclass of Dequeue.

If a child class overrides a method inherited from the parent in a way that makes it unusable (for example, issues an error message), then we call it inheritance for limitation.

For example, you have an existing **List** data type that allows items to be inserted at either end, and you override methods allowing insertion at one end in order to create a **Stack**.

Generally not a good idea, since it breaks the idea of substitution. But again, it is sometimes found in practice.

combination -- multiple inheritance. Provided in part by implementing multiple interfaces.

Benefits of Inheritance

- Software Reuse
- Code Sharing
- Improved Reliability
- Consistency of Interface
- Rapid Prototyping
- Polymorphism
- Information Hiding

Cost of Inheritance

- Execution speed
- Program size
- Message Passing Overhead
- Program Complexity

This does not mean you should not use inheritance, but rather than you must understand the benefits, and weigh the benefits against the costs.