Classes and Objects in Java

Classes and Objects are basic concepts of Object Oriented Programming that revolve around real-life entities.

Class

1. Class is a set of objects which shares common characteristics/ behaviour and common properties/ attributes.
2. Class is not a real-world entity. It is just a template or blueprint or prototype from which objects are created.
3. Class does not occupy memory.
4. Class is a group of variables of different data types and a group of methods.

A class in java can contain:

- data member
- method
- constructor
- nested class and
- interface

Syntax to declare a class:

access_modifier class<class_name>

{

data member;

method;

constructor;

nested class;

interface;

}

Example:

- Animal
- Student
- Bird
- Vehicle
- Company

```
class Student {
    int id; // data member (also instance variable)
    String name; // data member (also instance variable)
     public static void main(String args[])
    {
        Student s1 = new Student(); // creating an object of Student
        System.out.println(s1.id);
        System.out.println(s1.name);
    }
}
```

A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers**: A class can be public or has default
2. **Class keyword:** class keyword is used to create a class.
3. **Class name:** The name should begin with an initial letter (capitalized by convention).
4. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
5. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
6. **Body:** The class body is surrounded by braces, { }.

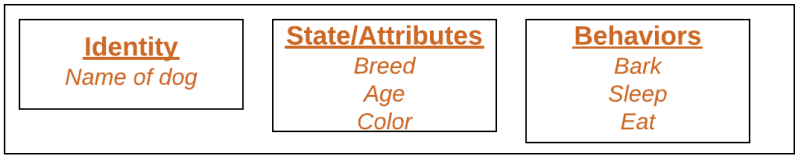
Constructors are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

Object

It is a basic unit of Object-Oriented Programming and represents real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :

1. **State**: It is represented by attributes of an object. It also reflects the properties of an object.
2. **Behaviour**: It is represented by the methods of an object. It also reflects the response of an object with other objects.
3. **Identity**: It gives a unique name to an object and enables one object to interact with other objects.

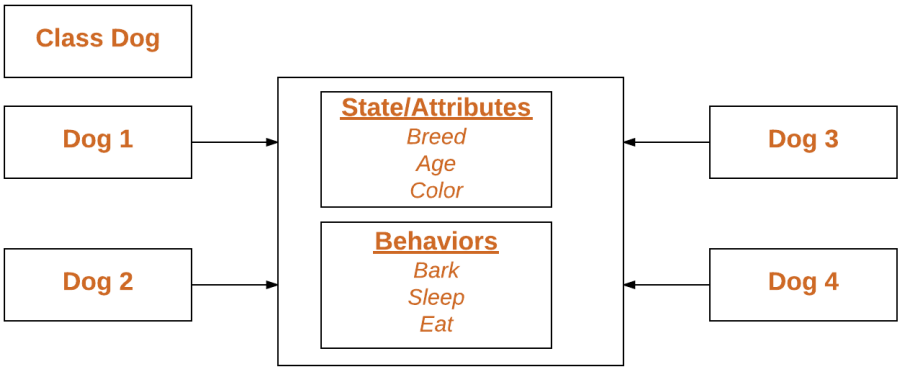
Example of an object: dog



Objects correspond to things found in the real world. For example, a graphics program may have objects such as "circle", "square", and "menu". An online shopping system might have objects such as "shopping cart", "customer", and "product".

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.
Example:



Ways to create an object of a class

Using new keyword: It is the most common and general way to create an object in java. Example:
// creating object of class Test & call show() method

```
Test t = new Test();
t.show();
```

Anonymous objects

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

- They are used for immediate method calls.
- They will be destroyed after method calling.

// creating object of class Test & call show() method

```
new Test().show();
```

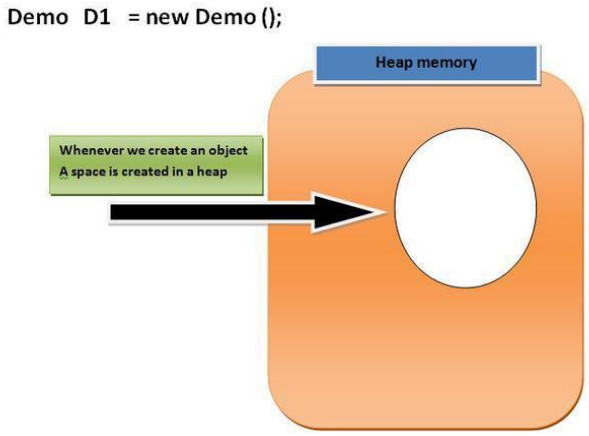
Difference between Class and Object are as follows:

Class	**Object**
Class is the blueprint of an object. It is used to create objects.	An object is an instance of the class.
No memory is allocated when a class is declared.	Memory is allocated as soon as an object is created.
A class is a group of similar objects.	An object is a real-world entity such as a book, car, etc.
Class is a logical entity.	An object is a physical entity.

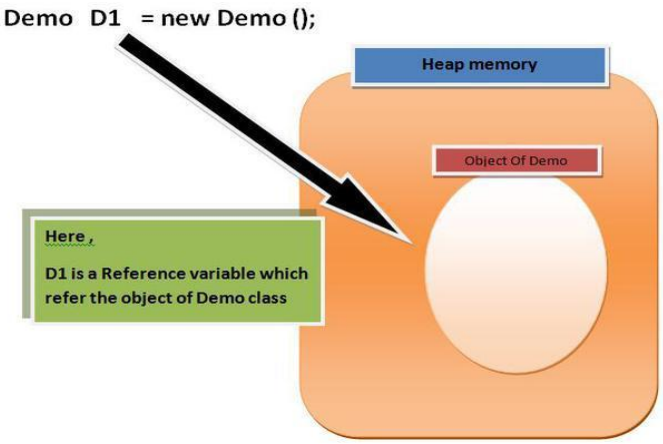
A class can only be declared once.	Objects can be created many times as per requirement.
An example of class can be a car.	Objects of the class car can be BMW, Mercedes, Ferrari, etc.

Reference Variable in Java

Before We Started with the **Reference variable** we should know about the following facts.
1. When we create an object (instance) of class then space is reserved in heap memory. Let's understand with the help of an example.
Demo D1 = new Demo();



Now, The space in the heap Memory is created but the question is **how to access that space?**.
Then, We create a Pointing element or simply called **Reference variable** which simply points out the Object(the created space in a Heap Memory).

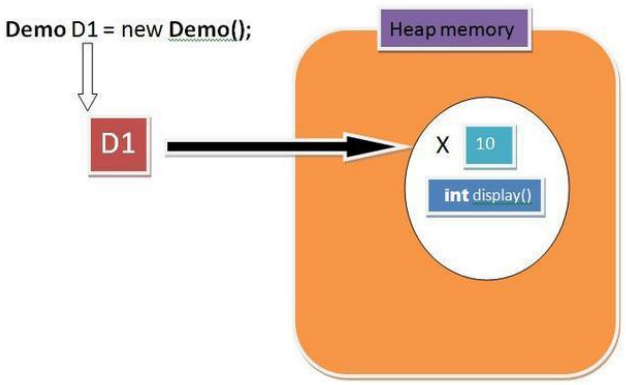


Understanding Reference variable
1. Reference variable is used to point object/values.
2. Classes, interfaces, arrays, enumerations, and, annotations are reference types in Java. Reference variables hold the objects/values of reference types in Java.
3. Reference variable can also store **null** value. By default, if no object is passed to a reference variable then it will store a null value.
4. You can access object members using a reference variable using **dot** syntax.
<reference variable name >.<instance variable_name / method_name>

```
import java.io.*;
class Demo {
    int x = 10;

    int display()
    {
        System.out.println("x = " + x);

    }
}

class Main {
    public static void main(String[] args)
    {
          // create instance
        Demo D1 = new Demo();

        // accessing instance(object) variable
        System.out.println(D1.x);

        // point 3
        // accessing instance(object) method
        D1.display();
    }
}
```

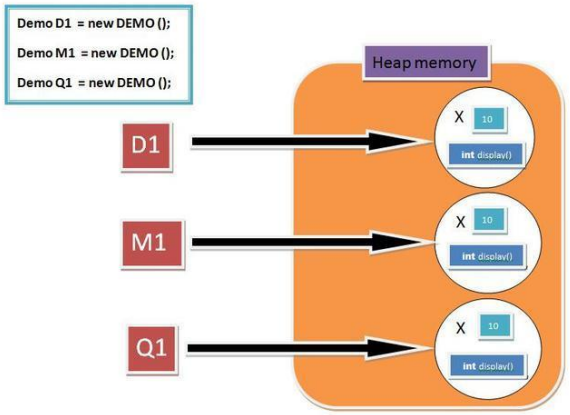
Output

```
10

x = 10
```




```
import java.io.*;
class Demo {

    int x = 10;
    int display()
    {
        System.out.println("x = " + x);
        return 0;
    }
}
class Main {
    public static void main(String[] args)
    {
          // create instances
        Demo D1 = new Demo();

        Demo M1 = new Demo();
```

```
        Demo Q1 = new Demo();
    }
}
```

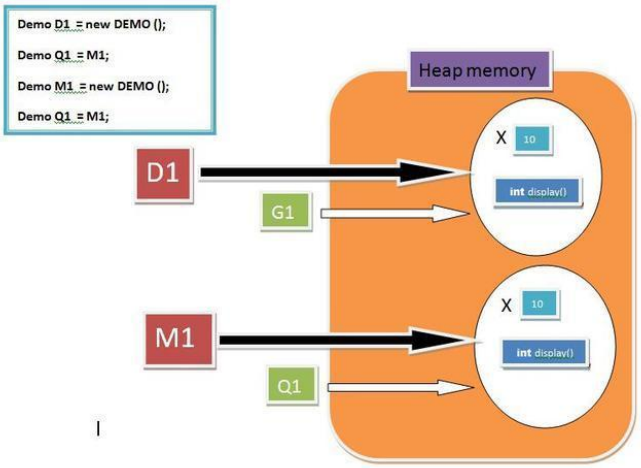



```
import java.io.*;
class Demo {
    int x = 10;
    int display()
    {
        System.out.println("x = " + x);
        return 0;
    }
}
class Main {
    public static void main(String[] args)
    {
        // create instance
        Demo D1 = new Demo();
        // point to same reference
        Demo G1 = D1;
        Demo M1 = new Demo();
        Demo Q1 = M1;
        // updating the value of x using G!
        // reference variable
        G1.x = 25;
        System.out.println(G1.x); // Point 1
        System.out.println(D1.x); // Point 2
    }
}
```

Output

25

25



Note:
Here we pass G1 and Q1 reference variable point out the same object respectively. Secondly ***At Point 1*** *we try to get the value of the object with G1 reference variable which shows it as* ***25*** *and* ***At Point 2*** *we try to get the value of an object with D1 reference variable which shows it as* ***25*** *as well. This will prove that the modification in the object can be done by using any reference variable but the condition is it should hold the same reference.*

Static & Non-Static

The main differences between static and non-static variables are:

Static variable	**Non static variable**
Static variables can be accessed using class name	Non static variables can be accessed using instance of a class
Static variables can be accessed by static and non static methods	Non static variables cannot be accessed inside a static method.
Static variables reduce the amount of memory used by a program.	Non static variables do not reduce the amount of memory used by a program
Static variables are shared among all instances of a class.	Non static variables are specific to that instance of a class.
Static variable is like a global variable and is available to all methods.	Non static variable is like a local variable and can be accessed through only instance of a class.

Examples are given in the notebook.

The main differences between static and non-static method are:

Points	Static method	Non-static method
Definition	A **static method** is a method that belongs to a class, but it does not belong to an instance of that class and this method can be called without the instance or object of that class.	Every method in java defaults to a non-static method without a **static** keyword preceding it. **non-static** methods can access any **static** method and **static** variable also, without using the object of the class.
Accessing members and methods	In the **static** method, the method can only access only static data members and static methods of another class or same class but cannot access non-static methods and variables.	In the **non-static** method, the method can access static data members and static methods as well as non-static members and methods of another class or same class.
Binding process	The static method uses compile-time or early binding.	The non-static method uses runtime or dynamic binding.
Overriding	The static method cannot be overridden because of early binding.	The non-static method can be overridden because of runtime binding.
Memory allocation	In the **static** method, less memory is used for execution because memory allocation happens only once because the static keyword fixed a particular memory for that method in ram.	In the **non-static** method, much memory is used for execution because here memory allocation happens when the method is invoked and the memory is allocated every time when the method is called.

Example is given in notebook.

Java Method Parameters

Information can be passed to methods as parameter. Parameters act as variables inside the method.

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a String called **fname** as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

```
public class Main {

  static void myMethod(String fname) {

    System.out.println(fname + " Refsnes");

  }
```

```
  public static void main(String[] args) {
    myMethod("Liam");
    myMethod("Jenny");
    myMethod("Anja");
  }
}
```

Multiple Parameters

You can have as many parameters as you like:

```
public class Main {
  static void myMethod(String fname, int age) {
    System.out.println(fname + " is " + age);
  }

  public static void main(String[] args) {
    myMethod("Liam", 5);
    myMethod("Jenny", 8);
    myMethod("Anja", 31);
  }
}

// Liam is 5
// Jenny is 8
// Anja is 31
```

Return Values

The void keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as int, char, etc.) instead of void, and use the return keyword inside the method:

Example

```
public class Main {
```

```
  static int myMethod(int x) {
    return 5 + x;
  }

  public static void main(String[] args) {
    int y= myMethod(3);
    System.out.println(y);
  }
}
// Outputs 8 (5 + 3)
```

This example returns the sum of a method's **two parameters**:

```
public class Main {
  static int myMethod(int x, int y) {
    return x + y;
  }

  public static void main(String[] args) {
    int y = myMethod(5, 3);
    System.out.println(y);
  }
}
// Outputs 8 (5 + 3)
```

Method Overloading

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 behaviour of the method because its name differs.

So, we perform method overloading to figure out the program quickly.

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

1) Method Overloading: changing no. of arguments

In this example, we have created two methods. The 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. }}
```

Output:

```
22
33
```

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. }}
```

Output:

```
22
24.9
```

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

Can we overload java main() method?

Yes, 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.  }
```

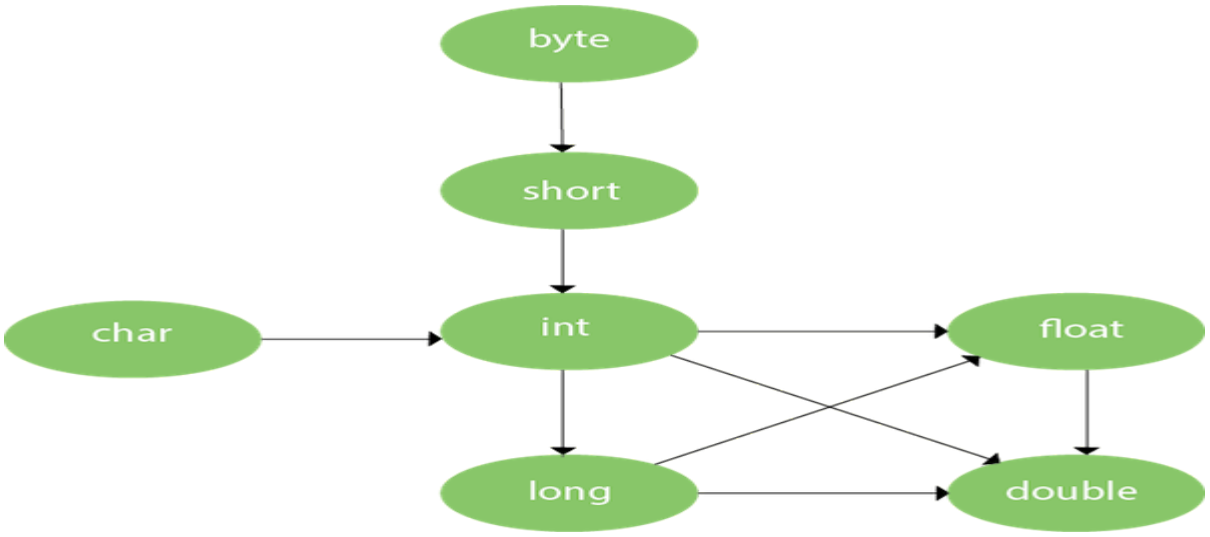
Output:

```
main with String[]
```

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:

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.
5.    public static void main(String args[]){
6.    OverloadingCalculation1 obj=new OverloadingCalculation1();
7.    obj.sum(20,20);//now second int literal will be promoted to long
8.    obj.sum(20,20,20);
9.
10.   }
11. }
```

```
Output:40
       60
```

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.
```

```
5.   public static void main(String args[]){
6.   OverloadingCalculation2 obj=new OverloadingCalculation2();
7.   obj.sum(20,20);//now int arg sum() method gets invoked
8.   }
9.  }
```

```
Output:int arg method invoked
```

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 a 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.
5.   public static void main(String args[]){
6.   OverloadingCalculation3 obj=new OverloadingCalculation3();
7.   obj.sum(20,20);//now ambiguity
8.   }
9.  }
```

```
Output:Compile Time Error
```

Java Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Recursion Example

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

Example

Use recursion to add all of the numbers up to 10.

```
public class Main {
  public static void main(String[] args) {
    int result = sum(10);
    System.out.println(result);
  }
  public static int sum(int k) {
    if (k > 0) {
      return k + sum(k - 1);
    } else {
      return 0;
    }
  }
}
```

Example Explained

When the `sum()` function is called, it adds parameter `k` to the sum of all numbers smaller than `k` and returns the result. When k becomes 0, the function just returns 0. When running, the program follows these steps:

```
10 + sum(9)
10 + ( 9 + sum(8) )
10 + ( 9 + ( 8 + sum(7) ) )
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
```

Since the function does not call itself when `k` is 0, the program stops there and returns the result.

Constructor

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 the 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, the 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 the Java compiler creates a default constructor if your class doesn't have any.

Rules for creating Java constructor

There are two rules defined for the constructor.

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

Types of Java constructors

There are two types of constructors in Java:

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

Java Default Constructor

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

Syntax of default constructor:

1. <class_name>(){}

Example of default constructor

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

```
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. }
10.}
```

Output:

```
Bike is created
```

Parameterized Constructor

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

Why use the parameterized constructor?

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

Example of parameterized constructor

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

```
1. //Java Program to demonstrate the use of the parameterized constructor.
2. class Student4{
3.     int id;
4.     String name;
5.     //creating a parameterized constructor
6.     Student4(int i,String n){
7.     id = i;
8.     name = n;
9.     }
10.    //method to display the values
11.    void display(){System.out.println(id+" "+name);}
12.
13.    public static void main(String args[]){
14.    //creating objects and passing values
15.    Student4 s1 = new Student4(111,"Karan");
16.    Student4 s2 = new Student4(222,"Aryan");
17.    //calling method to display the values of object
18.    s1.display();
19.    s2.display();
20.   }
21.}
```

Output:

```
111 Karan

222 Aryan
```

Constructor Overloading in Java

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 of Constructor Overloading

```
1.  //Java program to overload constructors
2.  class Student5{
3.      int id;
4.      String name;
5.      int age;
6.      //creating two arg constructor
7.      Student5(int i,String n){
8.      id = i;
9.      name = n;
10.     }
11.     //creating three arg constructor
12.     Student5(int i,String n,int a){
13.     id = i;
14.     name = n;
15.     age=a;
16.     }
17.     void display(){System.out.println(id+" "+name+" "+age);}
18.
19.     public static void main(String args[]){
20.     Student5 s1 = new Student5(111,"Karan");
21.     Student5 s2 = new Student5(222,"Aryan",25);
22.     s1.display();
23.     s2.display();
24.   }
25.}
```

Output:

```
111 Karan 0
222 Aryan 25
```

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.

Package in java can be categorized in two forms, built-in package and user-defined package.

There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.

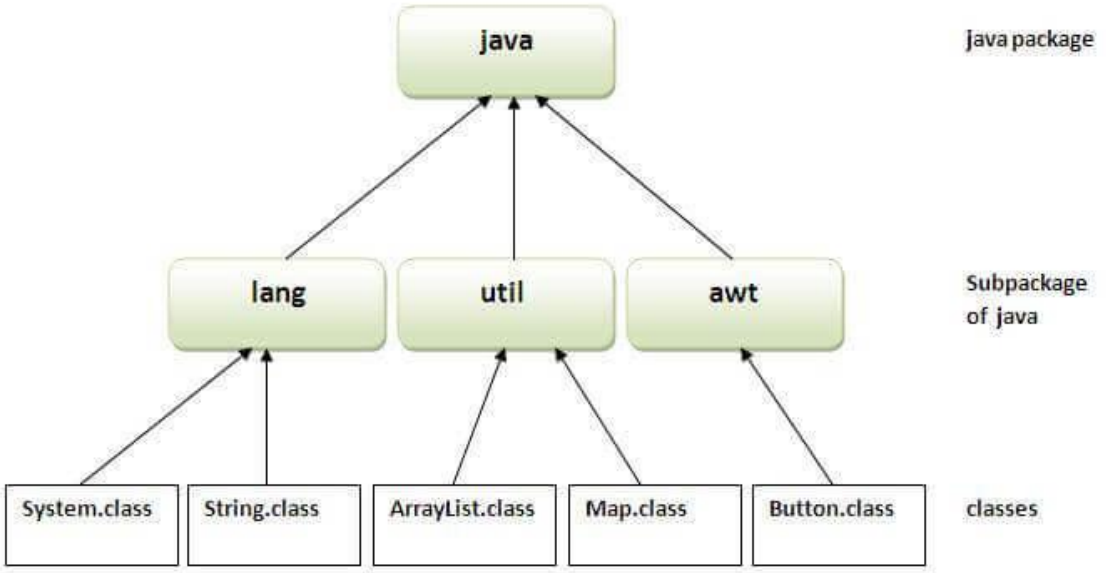
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.

2) Java package provides access protection.

3) Java package removes naming collisions.



Simple example of java package

The **package keyword** is used to create a package in java.

```
1. //save as Simple.java
2. package mypack;
3. public class Simple{
4.  public static void main(String args[]){
5.    System.out.println("Welcome to package");
6.  }
```

7. }

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

1. javac -d directory javafilename

For **example**

1. javac -d . Simple.java

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).

How to run java package program

You need to use a fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

Output: Welcome to package

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination. The . represents the current folder.

How to access a package from another package?

There are three ways to access the package from outside the package.

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not subpackages.

The import keyword is used to make the classes and interface of another package accessible to the current package.

Example of package that import the packagename.*

```
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.
5. class B{
6.   public static void main(String args[]){
7.    A obj = new A();
8.    obj.msg();
9.   }
10.}
```

Output:Hello

---

2) Using packagename.classname

If you import package.classname then only the declared class of this package will be accessible.

Example of package by import package.classname

```
1. //save by A.java
2.
3. package pack;
4. public class A{
5.   public void msg(){System.out.println("Hello");}
6. }
1. //save by B.java
2. package mypack;
3. import pack.A;
```

```
4.
5.  class B{
6.    public static void main(String args[]){
7.     A obj = new A();
8.     obj.msg();
9.    }
10.}
```

Output:Hello

---

3) Using fully qualified name

If you use a fully qualified name then only the declared class of this package will be accessible. Now there is no need to import. But you need to use a fully qualified name every time when you are accessing the class or interface.

It is generally used when two packages have the same class name e.g. java.util and java.sql packages contain Date class.

Example of package by import fully qualified name

```
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.  class B{
4.    public static void main(String args[]){
5.     pack.A obj = new pack.A();//using fully qualified name
6.     obj.msg();
7.    }
8.  }
```

Output:Hello

Note: If you import a package, subpackages will not be imported.

If you import a package, all the classes and interfaces of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Subpackage in java

Package inside the package is called the **subpackage**. It should be created **to categorize the package further**.

Let's take an example, Sun Microsystem has defined a package named java that contains many classes like System, String, Reader, Writer, Socket etc. These classes represent a particular group e.g. Reader and Writer classes are for Input/Output operation, Socket and ServerSocket classes are for networking etc and so on. So, Sun has subcategorized the java package into subpackages such as lang, net, io etc. and put the Input/Output related classes in io package, Server and ServerSocket classes in net packages and so on.

> The standard of defining a package is domain.company.package e.g. com.p1.bean or org.sssit.dao.

Example of Subpackage

```
1. package com.p1.core;
2. class Simple{
3.   public static void main(String args[]){
4.    System.out.println("Hello subpackage");
5.   }
6. }
```

To Compile: javac -d . Simple.java

To Run: java com.p1.core.Simple

```
Output:Hello subpackage
```

Input and output functions

Java I/O (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow.

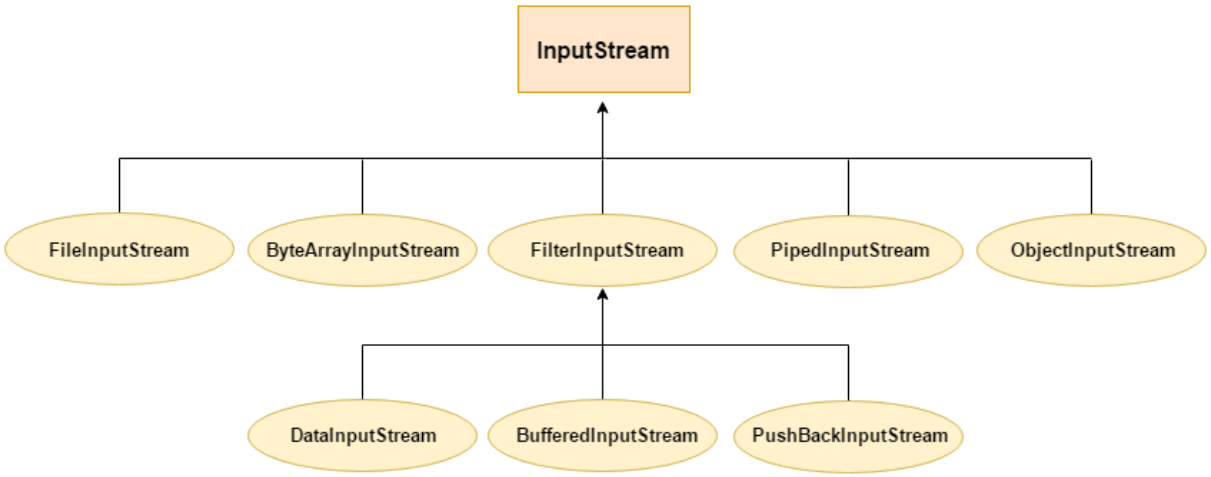
n Java, 3 streams are created for us automatically. All these streams are attached with the console.

1) System.out: standard output stream

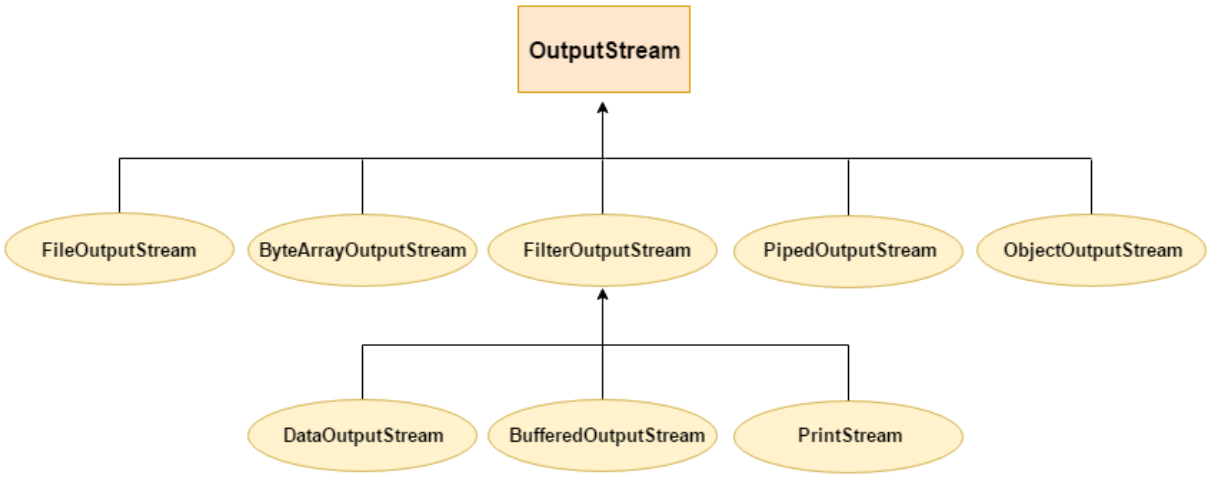
2) System.in: standard input stream

3) System.err: standard error stream

InputStream Hierarchy



OutputStream Hierarchy



Java Command Line Arguments

The java command-line argument is an argument i.e. passed at the time of running the java program.

The arguments passed from the console can be received in the java program and it can be used as an input.

So, it provides a convenient way to check the behavior of the program for the different values. You can pass **N** (1,2,3 and so on) numbers of arguments from the command prompt.

Simple example of command-line argument in java

In this example, we are receiving only one argument and printing it. To run this java program, you must pass at least one argument from the command prompt.

```
1. class CommandLineExample{
2. public static void main(String args[]){
3. System.out.println("Your first argument is: "+args[0]);
4. }
5. }
```

```
1. compile by > javac CommandLineExample.java
2. run by > java CommandLineExample sonoo
```

```
Output: Your first argument is: sonoo
```

Java Scanner

Scanner class in Java is found in the java.util package. Java provides various ways to read input from the keyboard, the java.util.Scanner class is one of them.

The Java Scanner class breaks the input into tokens using a delimiter which is whitespace by default. It provides many methods to read and parse various primitive values.

The Java Scanner class is widely used to parse text for strings and primitive types using a regular expression. It is the simplest way to get input in Java. By the help of Scanner in Java, we can get input from the user in primitive types such as int, long, double, byte, float, short, etc.

The Java Scanner class extends Object class and implements Iterator and Closeable interfaces.

The Java Scanner class provides nextXXX() methods to return the type of value such as nextInt(), nextByte(), nextShort(), next(), nextLine(), nextDouble(), nextFloat(), nextBoolean(), etc. To get a single character from the scanner, you can call next().charAt(0) method which returns a single character.

Java Scanner Class Declaration

```
1. public final class Scanner
```

2. **extends** Object
3. **implements** Iterator<String>

How to get Java Scanner

To get the instance of Java Scanner which reads input from the user, we need to pass the input stream (System.in) in the constructor of Scanner class. For Example:

```
1. Scanner in = new Scanner(System.in);
```

To get the instance of Java Scanner which parses the strings, we need to pass the strings in the constructor of Scanner class. For Example:

```
1. Scanner in = new Scanner("Hello Javatpoint");
```

Scanner Class Methods

Sr. No.	Return type	method name	Description
1	String	next()	It is used to get the next complete token from the scanner which is in use.
2	BigDecimal	nextBigDecimal()	It scans the next token of the input as a BigDecimal.
3	BigInteger	nextBigInteger()	It scans the next token of the input as a BigInteger.
4	boolean	nextBoolean()	It scans the next token of the input into a boolean value and returns that value.

5	byte	nextByte()	It scans the next token of the input as a byte.
6	double	nextDouble()	It scans the next token of the input as a double.
7	float	nextFloat()	It scans the next token of the input as a float.
8	int	nextInt()	It scans the next token of the input as an Int.
9	String	nextLine()	It is used to get the input string that was skipped of the Scanner object.
10	long	nextLong()	It scans the next token of the input as a long.
11	short	nextShort()	It scans the next token of the input as a short.

Example

Let's see a simple example of Java Scanner where we are getting a single input from the user. Here, we are asking for a string through the in.nextLine() method.

```
1. import java.util.*;
2. public class ScannerExample {
3. public static void main(String args[]){
4.         Scanner in = new Scanner(System.in);
5.         System.out.print("Enter your name: ");
```

```
6.          String name = in.nextLine();
7.          System.out.println("Name is: " + name);
8.          in.close();
9.          }
10.}
```

Output:

```
Enter your name: sonoo jaiswal
Name is: sonoo jaiswal
```