1. CLASSES, OBJECT AND METHODS:

Defining a Class

The first bold line in the following listing begins a *class definition block*.

```
/**
 * The HelloWorldApp class implements an application that
 * simply displays "Hello World!" to the standard output.
 */
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); //Display the string.
    }
}
```

A *class*--the basic building block of an object-oriented language such as Java--is a template that describes the data and behavior associated with *instances* of that class. When you *instantiate* a class you create an *object* that looks and feels like other instances of the same class. The data associated with a class or object is stored in *variables*; the behavior associated with a class or object is implemented with *methods*. Methods are similar to the functions or procedures in procedural languages such as C.

Julia Child's recipe for rack of lamb is a real-world example of a class. Her rendition of the rack of lamb is one instance of the recipe, and mine is quite another. (While both racks of lamb may "look and feel" the same, I imagine that they "smell and taste" different.)

A more traditional example from the world of programming is a class that represents a rectangle. The class would contain variables for the origin of the rectangle, its width, and its height. The class might also contain a method that calculates the area of the rectangle. An instance of the rectangle class would contain the information for a specific rectangle, such as the dimensions of the floor of your office, or the dimensions of this page.

In the Java language, the simplest form of a class definition is

```
class name {
    . . .
}
```

The keyword `class` begins the class definition for a class named `name`. The variables and methods of the class are embraced by the curly brackets that begin and end the class definition block. The "Hello World" application has no variables and has a single method named `main`.

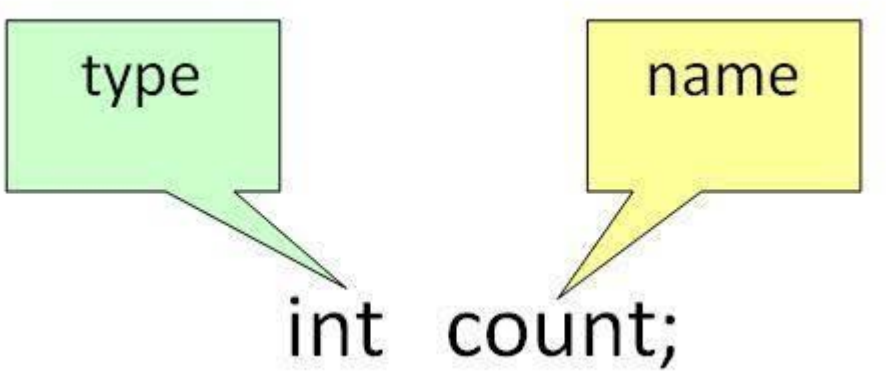
Variables in Java

A variable is a name given to a memory location. It is the basic unit of storage in a program.

- The value stored in a variable can be changed during program execution.
- A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.
- In Java, all the variables must be declared before use.

How to declare variables?

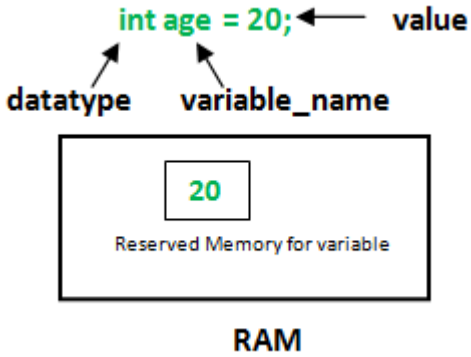
We can declare variables in java as follows:



type: Type of data that can be stored in this variable.
name: Name given to the variable.
In this way, a name can only be given to a memory location. It can be assigned values in two ways:

- Variable Initialization
- Assigning value by taking input

How to initialize variables?



datatype: Type of data that can be stored in this variable.
variable_name: Name given to the variable.
value: It is the initial value stored in the variable.
Examples:

```
float simpleInterest; //Declaring float variable

int time = 10, speed = 20; //Declaring and Initializing integer
variable
```

```
char var = 'h'; // Declaring and Initializing character variable
```

Types of variables

There are three types of variables in Java:

- Local Variables
- Instance Variables
- Static Variables

(Note: Unit 1.3 for referring about the types of variables in details.)

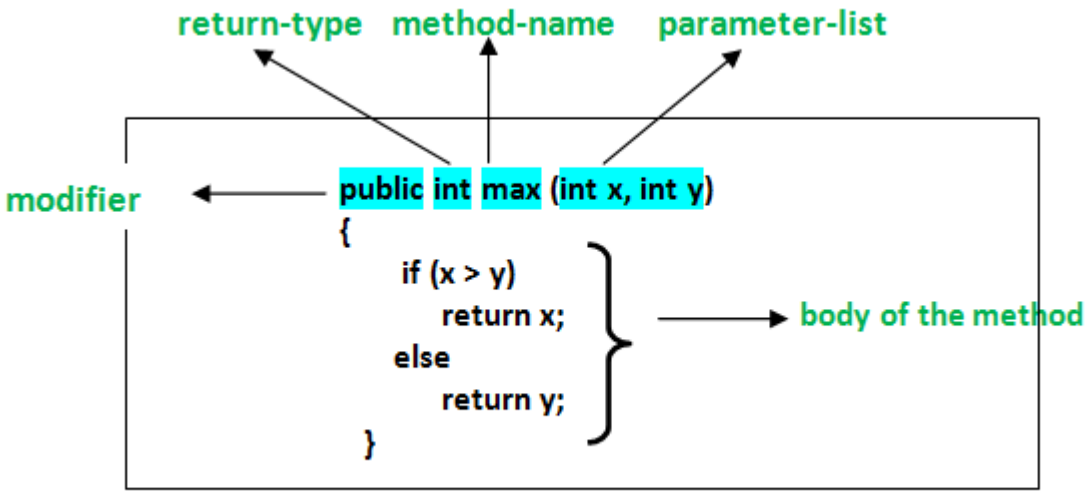
Methods in Java

A method is a collection of statements that perform some specific task and return the result to the caller. A method can perform some specific task without returning anything. Methods allow us to **reuse** the code without retyping the code. In Java, every method must be part of some class which is different from languages like C, C++, and Python.
Methods are **time savers** and help us to **reuse** the code without retyping the code.

Method Declaration

In general, method declarations has six components :

- **Modifier**-: Defines **access type** of the method i.e. from where it can be accessed in your application. In Java, there 4 type of the access specifiers.
  - public: accessible in all class in your application.
  - protected: accessible within the class in which it is defined and in its **subclass(es)**
  - private: accessible only within the class in which it is defined.
  - default (declared/defined without using any modifier) : accessible within same class and package within which its class is defined.
- **The return type** : The data type of the value returned by the method or void if does not return a value.
- **Method Name** : the rules for field names apply to method names as well, but the convention is a little different.
- **Parameter list** : Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().
- **Exception list** : The exceptions you expect by the method can throw, you can specify these exception(s).
- **Method body** : it is enclosed between braces. The code you need to be executed to perform your intended operations.



Method signature: It consists of the method name and a parameter list (number of parameters, type of the parameters and order of the parameters). The return type and exceptions are not considered as part of it.
Method Signature of above function:

```
max(int x, int y)
```

How to name a Method?: A method name is typically a single word that should be a **verb** in lowercase or multi-word, that begins with a **verb** in lowercase followed by **adjective, noun…..** After the first word, first letter of each word should be capitalized. For example, findSum,
computeMax, setX and getX
Generally, A method has a unique name within the class in which it is defined but sometime a method might have the same name as other method names within the same class as method overloading is allowed in Java.

Calling a method

The method needs to be called for using its functionality. There can be three situations when a method is called:
A method returns to the code that invoked it when:

- It completes all the statements in the method
- It reaches a return statement
- Throws an exception

```
// Program to illustrate methodsin java
import java.io.*;

class Addition {

    int sum = 0;

    public int addTwoInt(int a, int b){

        // adding two integer value.
        sum = a + b;

```

```
        //returning summation of two values.
        return sum;
    }

}

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

        // creating an instance of Addition class
        Addition add = new Addition();

        // calling addTwoInt() method to add two integer using instance created
        // in above step.
        int s = add.addTwoInt(1,2);
        System.out.println("Sum of two integer values :"+ s);

    }
}
```

Output :

```
Sum of two integer values :3
```

See the below example to understand method call in detail :

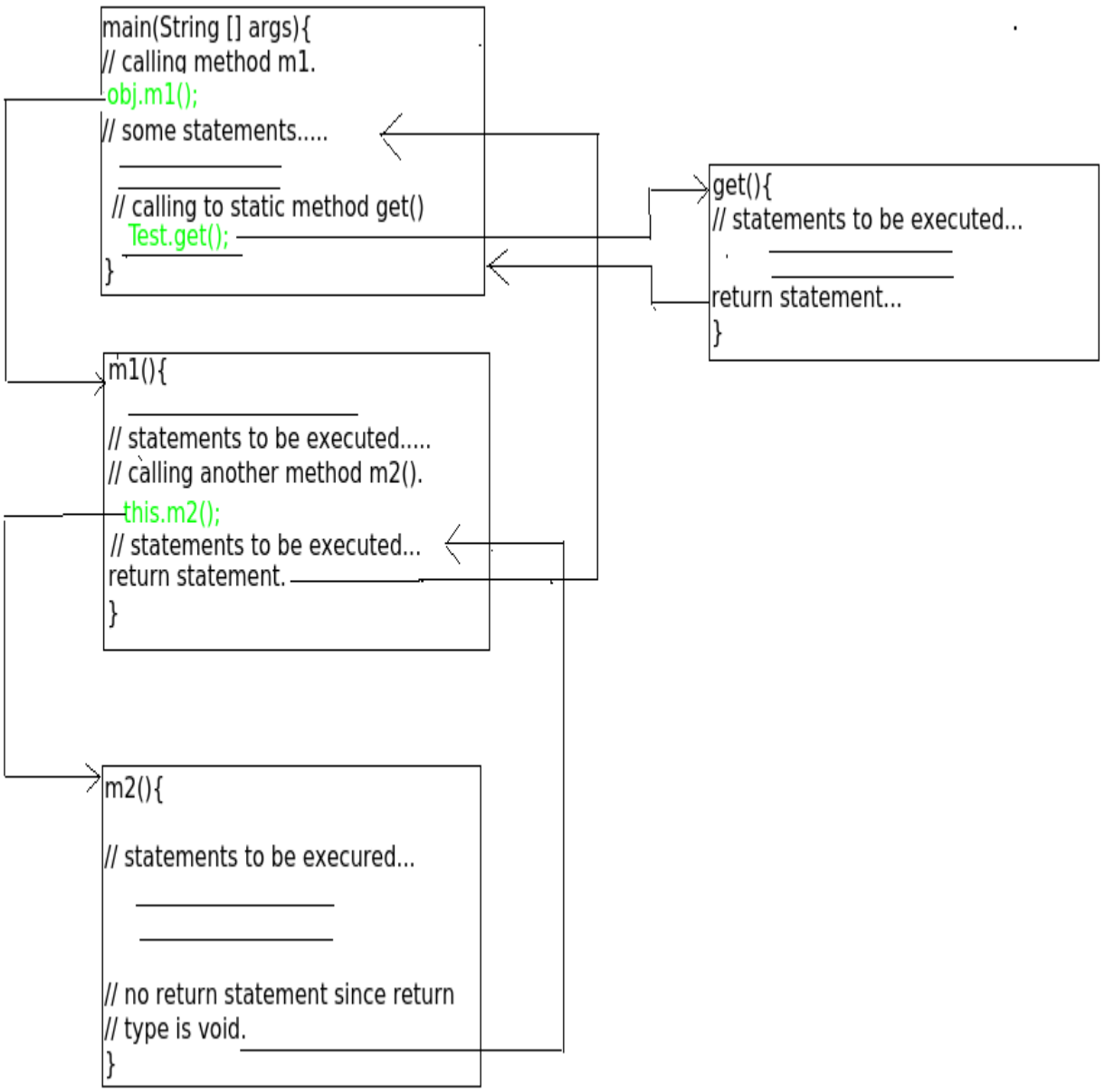
```
// Java program to illustrate different ways of calling a method
import java.io.*;

class Test
{
    public static int i = 0;
    // constructor of class which counts
    //the number of the objects of the class.
    Test()
    {
        i++;

    }
    // static method is used to access static members of the class
    // and for getting total no of objects
    // of the same class created so far
    public static int get()
    {
        // statements to be executed....
        return i;
    }

    // Instance method calling object directly
    // that is created inside another class 'GFG'.
    // Can also be called by object directly created in the same class
    // and from another method defined in the same class
    // and return integer value as return type is int.
    public int m1()
    {
        System.out.println("Inside the method m1 by object of GFG class");
```

```
        // calling m2() method within the same class.
        this.m2();

        // statements to be executed if any
        return 1;
    }

    // It doesn't return anything as
    // return type is 'void'.
    public void m2()
    {

        System.out.println("In method m2 came from method m1");
    }
}

class GFG
{
    public static void main(String[] args)
    {
        // Creating an instance of the class
        Test obj = new Test();

        // Calling the m1() method by the object created in above step.
        int i = obj.m1();
        System.out.println("Control returned after method m1 :" + i);

        // Call m2() method
        // obj.m2();
        int no_of_objects = Test.get();

        System.out.print("No of instances created till now : ");
        System.out.println(no_of_objects);

    }
}
```

Output :

```
Inside the method m1 by object of GFG class
In method m2 came from method m1
Control returned after method m1 :1
No of instances created till now : 1
```

Control flow of above program:



Memory allocation for methods calls

Methods calls are implemented through stack. Whenever a method is called a stack frame is created within the stack area and after that the arguments passed to and the local variables and value to be returned by this called method are stored in this stack frame and when execution of the called method is finished, the allocated stack frame would be deleted. There is a stack pointer register that tracks the top of the stack which is adjusted accordingly.

Different ways to create objects in Java

As you all know, in Java, a class provides the blueprint for objects, you create an object from a class. There are many different ways to create objects in Java.
Following are some ways in which you can create objects in Java:
1) Using new Keyword : Using new keyword is the most basic way to create an object. This is the most common way to create an object in java. Almost 99% of objects are created in this way. By using this method we can call any constructor we want to call (no argument or parameterized constructors).

```
// Java program to illustrate creation of Object
// using new keyword
public class NewKeywordExample
{
    String name = "CodeForIT";
    public static void main(String[] args)
    {
        // Here we are creating Object of
        // NewKeywordExample using new keyword
        NewKeywordExample obj = new NewKeywordExample();
        System.out.println(obj.name);
    }
}
```

Output:

```
CodeForIT
```

2) Using New Instance : If we know the name of the class & if it has a public default constructor we can create an object –**Class.forName**. We can use it to create the Object of a Class. Class.forName actually loads the Class in Java but doesn't create any Object. To Create an Object of the Class you have to use the new Instance Method of the Class.

```
// Java program to illustrate creation of Object
// using new Instance
public class NewInstanceExample
{
    String name = "CodeForIT";
    public static void main(String[] args)
    {
        try
        {
            Class cls = Class.forName("NewInstanceExample");
            NewInstanceExample obj =
                    (NewInstanceExample) cls.newInstance();
            System.out.println(obj.name);
        }
        catch (ClassNotFoundException e)
        {
            e.printStackTrace();
        }
        catch (InstantiationException e)
        {
            e.printStackTrace();
        }
        catch (IllegalAccessException e)
        {
```

```
            e.printStackTrace();
        }
    }
}
```

Output:

```
CodeForIT
```

Accessing class members

Access Modifiers in Java

As the name suggests access modifiers in Java helps to restrict the scope of a class, constructor , variable , method or data member. There are four types of access modifiers available in java:

1. Default – No keyword required
2. Private
3. Protected
4. Public

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

1.Default: When no access modifier is specified for a class , method or data member – It is said to be having the **default** access modifier by default.

- The data members, class or methods which are not declared using any access modifiers i.e. having default access modifier are accessible **only within the same package**.

In this example, we will create two packages and the classes in the packages will be having the default access modifiers and we will try to access a class from one package from a class of second package.

```
//Java program to illustrate default modifier
package p1;

//Class Geeks is having Default access modifier
class Geek
{
    void display()
       {
           System.out.println("Hello World!");
       }
```

```
}
//Java program to illustrate error while
//using class from different package with
//default modifier
package p2;
import p1.*;

//This class is having default access modifier
class GeekNew
{
    public static void main(String args[])
       {
          //accessing class Geek from package p1
          Geeks obj = new Geek();

          obj.display();
       }
}
```

Output:

```
Compile time error
```

Private: The private access modifier is specified using the keyword **private**.

- The methods or data members declared as private are accessible only **within the class** in which they are declared.
- Any other **class of same package will not be able to access** these members.
- Top level Classes or interface can not be declared as private because
  1. private means "only visible within the enclosing class".
  2. protected means "only visible within the enclosing class and any subclasses"

  Hence these modifiers in terms of application to classes, they apply only to nested classes and not on top level classes

In this example, we will create two classes A and B within same package p1. We will declare a method in class A as private and try to access this method from class B and see the result.

```
//Java program to illustrate error while
//using class from different package with
//private modifier
package p1;

class A
{
   private void display()
    {
        System.out.println("CodeForIT");
    }
}

class B
{
   public static void main(String args[])
      {
          A obj = new A();
```

```
            //trying to access private method of another class
            obj.display();
        }
}
```

Output:

```
error: display() has private access in A
        obj.display();
```

protected: The protected access modifier is specified using the keyword **protected**.

- The methods or data members declared as protected are **accessible within same package or sub classes in different package.**

In this example, we will create two packages p1 and p2. Class A in p1 is made public, to access it in p2. The method display in class A is protected and class B is inherited from class A and this protected method is then accessed by creating an object of class B.

```
//Java program to illustrate
//protected modifier
package p1;

//Class A
public class A
{
   protected void display()
    {
        System.out.println("CodeForIT");
    }
}
//Java program to illustrate
//protected modifier
package p2;
import p1.*; //importing all classes in package p1

//Class B is subclass of A
class B extends A
{
   public static void main(String args[])
   {
       B obj = new B();
       obj.display();
   }

}
```

Output:

```
CodeForIT
```

public: The public access modifier is specified using the keyword **public**.

- The public access modifier has the **widest scope** among all other access modifiers.
- Classes, methods or data members which are declared as public are **accessible from every where** in the program. There is no restriction on the scope of a public data members.
  - `//Java program to illustrate`

```
//public modifier
package p1;
public class A
{
   public void display()
      {
          System.out.println("CodeForIT");
      }
}
package p2;
import p1.*;
class B
{
    public static void main(String args[])
      {
          A obj = new A;
          obj.display();
      }
}
```

Output:

```
CodeForIT
```

Constructors in Java

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

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

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

Constructors in Java

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

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

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

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

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

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

Rules for creating Java constructor

There are two rules defined for the constructor.

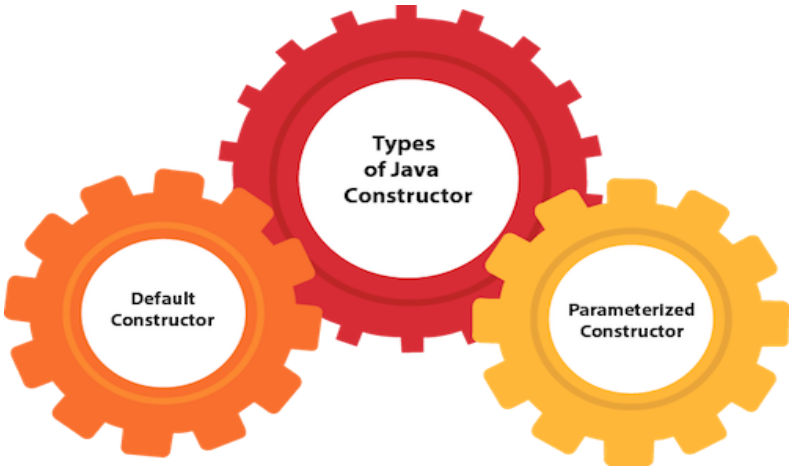
1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type
3. A Java constructor cannot be abstract, static, final, and synchronized

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



Constructors in Java

In Java, a constructor is a block of codes similar to the method.

It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

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

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

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

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

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

Rules for creating Java constructor

There are two rules defined for the constructor.

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

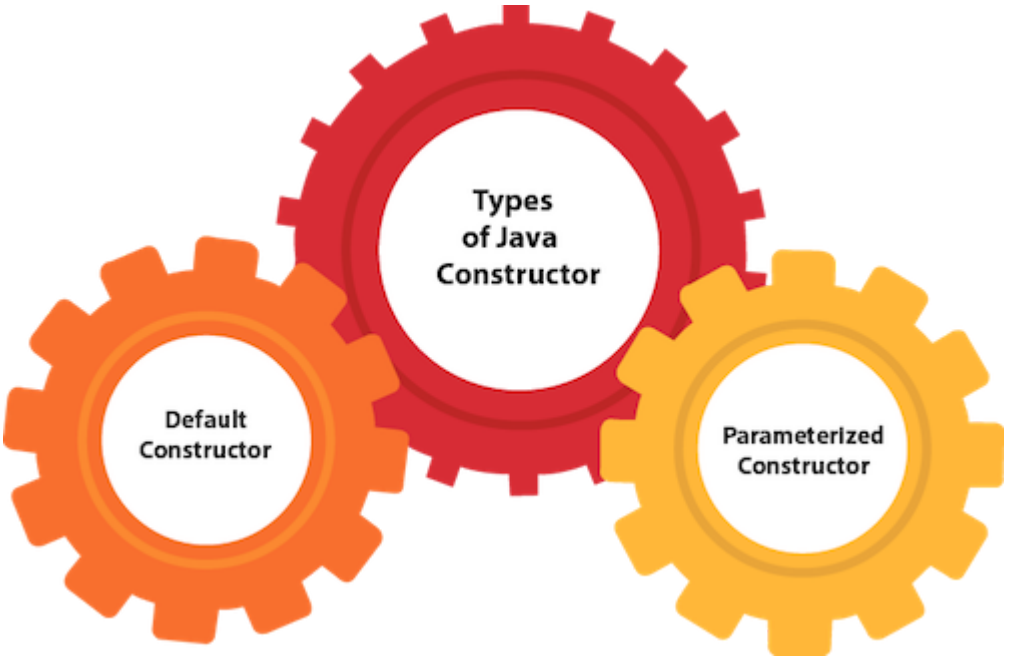
---

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



In this example, we are creating the no-arg constructor in the Bike class.

It will be invoked at the time of object creation.

Method Overloading in Java

Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or both. Overloading is related to compile-time (or static) polymorphism.

```
// Java program to demonstrate working of method
// overloading in Java.

public class Sum {

    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y)
    {
        return (x + y);
    }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z)
```

```
    {
        return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }

    // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Output :

```
30

60

31.0
```

1. **Question Arises:**
   Q. What if the exact prototype does not match with arguments.
   Ans.
   Priority wise, compiler take these steps: Type Conversion but to higher type(in terms of range) in same family.
2. Type conversion to next higher family(suppose if there is no long data type available for an int data type, then it will search for the float data type).

Let's take an example to clear the concept:-

```
class Demo {
    public void show(int x)
    {
        System.out.println("In int" + x);
    }
    public void show(String s)
    {
        System.out.println("In String" + s);
    }
    public void show(byte b)
    {
        System.out.println("In byte" + b);
    }
}
class UseDemo {
    public static void main(String[] args)
    {
        byte a = 25;
        Demo obj = new Demo();
        obj.show(a); // it will go to
```

```
        // byte argument
        obj.show("hello"); // String
        obj.show(250); // Int
        obj.show('A'); // Since char is
        // not available, so the datatype
        // higher than char in terms of
        // range is int.
        obj.show("A"); // String
        obj.show(7.5); // since float datatype
// is not available and so it's higher
// datatype, so at this step their
// will be an error.
}
}
```

What is the advantage?

We don't have to create and remember different names for functions doing the same thing. For example, in our code, if overloading was not supported by Java, we would have to create method names like sum1, sum2, … or sum2Int, sum3Int, … etc.

Can we overload methods on return type?

We **cannot** overload by return type. This behavior is same in C++. Refer this for details

```
public class Main {
    public int foo() { return 10; }

    // compiler error: foo() is already defined
    public char foo() { return 'a'; }

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

However, Overloading methods on return type are possible in cases where the data type of the function being called is explicitly specified. Look at the examples below :

```
// Java program to demonstrate the working of method
// overloading in static methods
public class Main {

    public static int foo(int a) { return 10; }
    public static char foo(int a, int b) { return 'a'; }

    public static void main(String args[])
    {
        System.out.println(foo(1));
        System.out.println(foo(1, 2));
    }
}
```

Output:

```
10

a
```

```
// Java program to demonstrate working of method
// overloading in  methods
class A {
```

```
    public int foo(int a) { return 10; }

    public char foo(int a, int b) { return 'a'; }
}

public class Main {

    public static void main(String args[])
    {
        A a = new A();
        System.out.println(a.foo(1));
        System.out.println(a.foo(1, 2));
    }
}
```

Output:

```
10

a
```

Can we overload static methods?

The answer is '**Yes**'. We can have two ore more static methods with same name, but differences in input parameters. For example, consider the following Java program. Refer this for details.

Can we overload methods that differ only by static keyword?

We **cannot** overload two methods in Java if they differ only by static keyword (number of parameters and types of parameters is same). See following Java program for example. Refer this for details.

Can we overload main() in Java?

Like other static methods, we **can** overload main() in Java. Refer overloading main() in Java for more details.

```
// A Java program with overloaded main()
import java.io.*;

public class Test {

    // Normal main()
    public static void main(String[] args)
    {
        System.out.println("Hi Geek (from main)");
        Test.main("Geek");
    }

    // Overloaded main methods
    public static void main(String arg1)
    {
        System.out.println("Hi, " + arg1);
        Test.main("Dear Geek", "My Geek");
    }
    public static void main(String arg1, String arg2)
    {
        System.out.println("Hi, " + arg1 + ", " + arg2);
    }
}
```

Output :

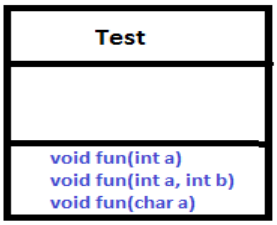
```
Hi Geek (from main)

Hi, Geek

Hi, Dear Geek, My Geek
```

Does Java support Operator Overloading?

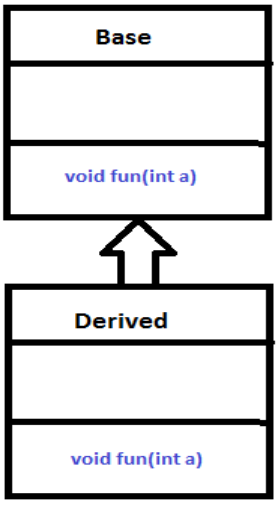
Unlike C++, Java doesn't allow user-defined overloaded operators. Internally Java overloads operators, for example, + is overloaded for concatenation.

What is the difference between Overloading and Overriding?

- Overloading is about same function have different signatures. Overriding is about same function, same signature but different classes connected through inheritance.



Overloading



Overriding

- Overloading is an example of compiler time polymorphism and overriding is an example of run time polymorphism.
- In Java, static members are those which belongs to the class and you can access these members without instantiating the class.
- The static keyword can be used with methods, fields, classes (inner/nested), blocks.
- **Static Methods** − You can create a static method by using the keyword *static*. Static methods can access only static fields, methods. To access static methods there is no need to instantiate the class, you can do it just using the class name as −
- **Example**

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

Output

```
Hello
```

Static Fields − You can create a static field by using the keyword static. The static fields have the same value in all the instances of the class. These are created and initialized when the class is loaded for the first time. Just like static methods you can access static fields using the class name (without instantiation).

Example

```
public class MyClass {

   public static int data = 20;

   public static void main(String args[]){

      System.out.println(MyClass.data);

   }

   Java Arrays with Answers

   27

}
```

Output

```
20
```

Static Blocks − These are a block of codes with a static keyword. In general, these are used to initialize the static members. JVM executes static blocks before the main method at the time of class loading.

Example

```
public class MyClass {

   static{

      System.out.println("Hello this is a static block");

   }

   public static void main(String args[]){

      System.out.println("This is main method");

   }

}
```

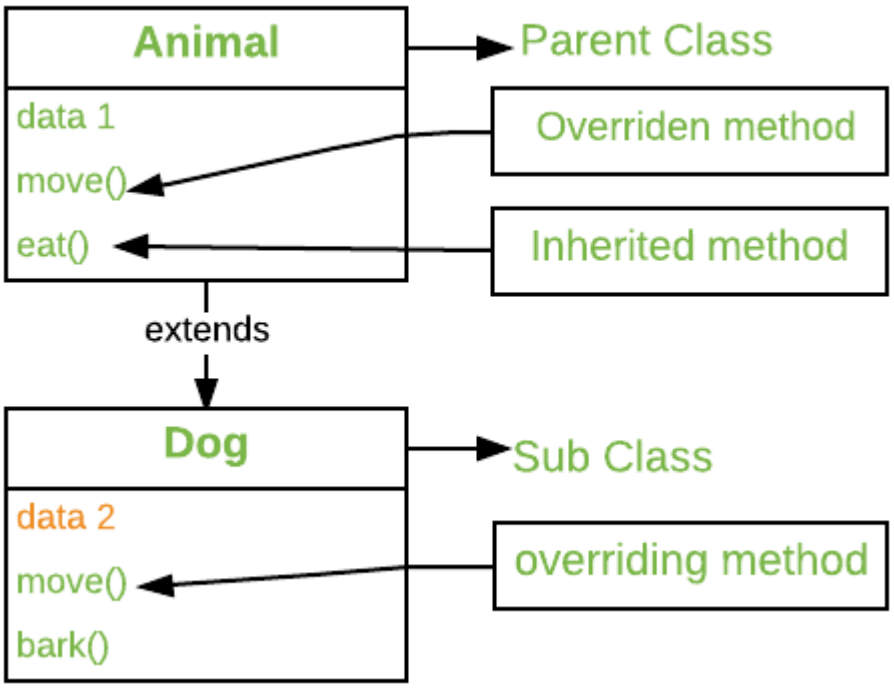
Output

```
Hello this is a static block
```

```
This is main method
```

Overriding in Java

In any object-oriented programming language, Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to *override* the method in the super-class.



Method overriding is one of the way by which java achieve Run Time Polymorphism.The version of a method that is executed will be determined by the object that is used to invoke it. If an object of a parent class is used to invoke the method, then the version in the parent class will be executed, but if an object of the subclass is used to invoke the method, then the version in the child class will be executed. In other words, *it is the type of the object being referred to* (not the type of the reference variable) that determines which version of an overridden method will be executed.

```
// A Simple Java program to demonstrate

// method overriding in java



// Base Class

class Parent {

    void show()
```

```
    {
        System.out.println("Parent's show()");
    }
}

// Inherited class
class Child extends Parent {
    // This method overrides show() of Parent
    @Override
    void show()
    {
        System.out.println("Child's show()");
    }
}

// Driver class
class Main {
    public static void main(String[] args)
    {
        // If a Parent type reference refers
        // to a Parent object, then Parent's
        // show is called
        Parent obj1 = new Parent();
        obj1.show();

        // If a Parent type reference refers
        // to a Child object Child's show()
        // is called. This is called RUN TIME
        // POLYMORPHISM.
```

```
        Parent obj2 = new Child();

        obj2.show();

    }

}
```

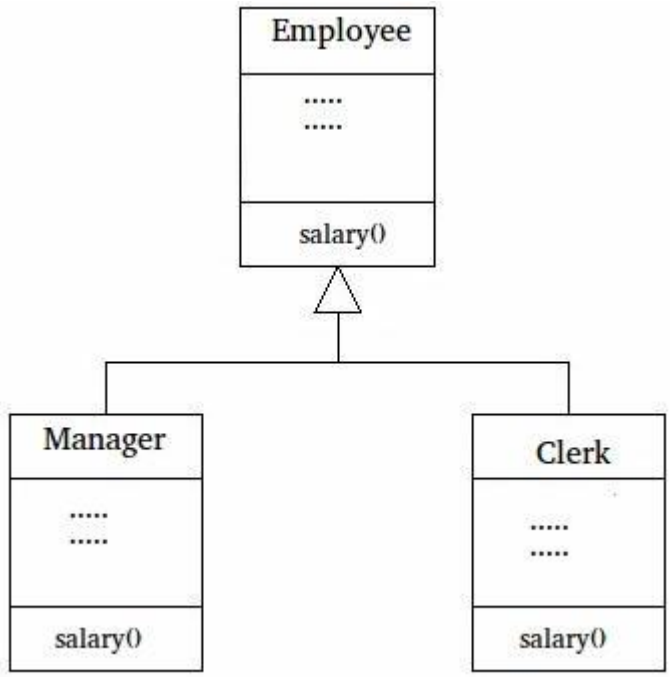
Output:

```
Parent's show()

Child's show()
```

When to apply Method Overriding ?(with example)

Overriding and Inheritance : Part of the key to successfully applying polymorphism is understanding that the superclasses and subclasses form a hierarchy which moves from lesser to greater specialization. Used correctly, the superclass provides all elements that a subclass can use directly. It also defines those methods that the derived class must implement on its own. This allows the subclass the flexibility to define its methods, yet still enforces a consistent interface. **Thus, by combining inheritance with overridden methods, a superclass can define the general form of the methods that will be used by all of its subclasses.**

Let's look at a more practical example that uses method overriding. Consider an employee management software for an organization, let the code has a simple base class Employee, the class has methods like raiseSalary(), transfer(), promote(), .. etc. Different types of employees like Manager, Engineer, ..etc may have their implementations of the methods present in base class Employee. In our complete software, we just need to pass a list of employees everywhere and call appropriate methods without even knowing the type of employee. For example, we can easily raise the salary of all employees by iterating through the list of employees. Every type of employee may have its logic in its class, we don't need to worry because if raiseSalary() is present for a specific employee type, only that method would be called.




```
// A Simple Java program to demonstrate application
// of overriding in Java

// Base Class
class Employee {
    public static int base = 10000;
    int salary()
    {
        return base;
    }
}

// Inherited class
class Manager extends Employee {
    // This method overrides salary() of Parent
    int salary()
    {
        return base + 20000;
    }
}

// Inherited class
class Clerk extends Employee {
    // This method overrides salary() of Parent
    int salary()
    {
        return base + 10000;
    }
}

// Driver class
class Main {
    // This method can be used to print the salary of
```

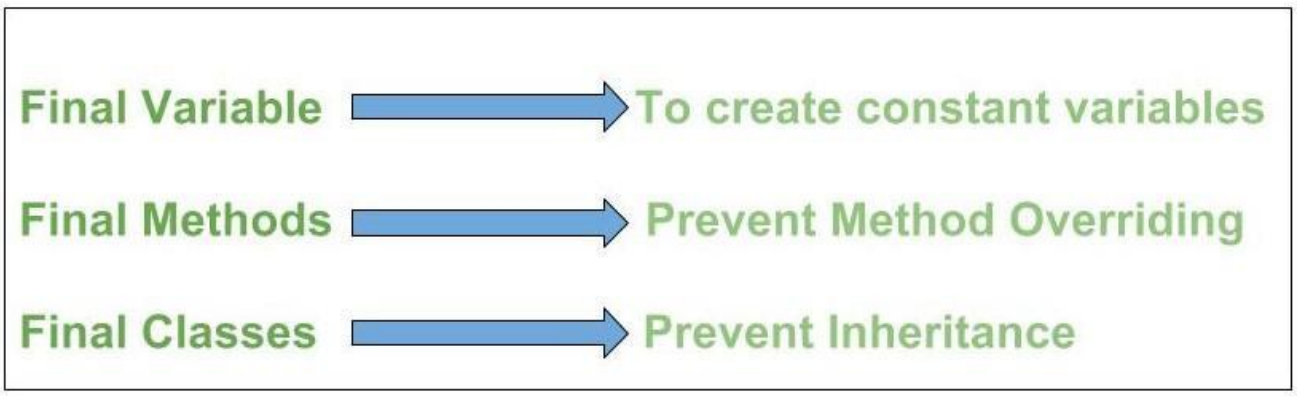
```
    // any type of employee using base class reference
    static void printSalary(Employee e)
    {
        System.out.println(e.salary());
    }

    public static void main(String[] args)
    {
        Employee obj1 = new Manager();

        // We could also get type of employee using
        // one more overridden method.loke getType()
        System.out.print("Manager's salary : ");
        printSalary(obj1);

        Employee obj2 = new Clerk();
        System.out.print("Clerk's salary : ");
        printSalary(obj2);
    }
}
```

Output:

```
Manager's salary : 30000

Clerk's salary : 20000
```

final keyword in java

final keyword is used in different contexts. First of all, *final* is a non-access modifier applicable **only to a variable, a method or a class**.Following are different contexts where final is used.



Example

Final variables

When a variable is declared with *final* keyword, its value can't be modified, essentially, a constant. This also means that you must initialize a final variable. If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from final array or final collection. It is good practice to represent final variables in all uppercase, using underscore to separate words.

Examples :

```
// a final variable
final int THRESHOLD = 5;
// a blank final variable
final int THRESHOLD;
// a final static variable PI
static final double PI = 3.141592653589793;
// a  blank final static  variable
static final double PI;
```

Initializing a final variable :

We must initialize a final variable, otherwise compiler will throw compile-time error.A final variable can only be initialized once, either via an initializer or an assignment statement. There are three ways to initialize a final variable :

1. You can initialize a final variable when it is declared.This approach is the most common. A final variable is called **blank final variable**,if it is **not** initialized while declaration. Below are the two ways to initialize a blank final variable.
2. A blank final variable can be initialized inside instance-initializer block or inside constructor. If you have more than one constructor in your class then it must be initialized in all of them, otherwise compile time error will be thrown.
3. A blank final static variable can be initialized inside static block.

Let us see above different ways of initializing a final variable through an example.

```
//Java program to demonstrate different
// ways of initializing a final variable

class Gfg
{
    // a final variable
    // direct initialize
    final int THRESHOLD = 5;

    // a blank final variable
    final int CAPACITY;

    // another blank final variable
    final int MINIMUM;

    // a final static variable PI
    // direct initialize
    static final double PI = 3.141592653589793;

    // a  blank final static  variable
    static final double EULERCONSTANT;

    // instance initializer block for
    // initializing CAPACITY
    {
        CAPACITY = 25;
    }
```

```
    // static initializer block for
    // initializing EULERCONSTANT
    static{
        EULERCONSTANT = 2.3;
    }

    // constructor for initializing MINIMUM
    // Note that if there are more than one
    // constructor, you must initialize MINIMUM
    // in them also
    public GFG()
    {
        MINIMUM = -1;
    }

}
```

When to use a final variable :

The only difference between a normal variable and a final variable is that we can re-assign value to a normal variable but we cannot change the value of a final variable once assigned. Hence final variables must be used only for the values that we want to remain constant throughout the execution of program.

Final classes

When a class is declared with *final* keyword, it is called a final class. A final class cannot be extended(inherited). There are two uses of a final class :

1. One is definitely to prevent inheritance, as final classes cannot be extended. For example, all Wrapper Classes like Integer,Float etc. are final classes. We can not extend them.

```
2. final class A
3. {
4.       // methods and fields
5. }
6. // The following class is illegal.
7. class B extends A
8. {
9.      // COMPILE-ERROR! Can't subclass A
10.   }
```

11. The other use of final with classes is to create an immutable class like the predefined String class.You can not make a class immutable without making it final.

Final methods

When a method is declared with *final* keyword, it is called a final method. A final method cannot be overridden. The Object class does this—a number of its methods are final.We must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes. The following fragment illustrates final keyword with a method:

```
class A
{
```

```
    final void m1()
    {
        System.out.println("This is a final method.");
    }
}

class B extends A
{
    void m1()
    {
        // COMPILE-ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

Java Object finalize() Method

Finalize() is the method of Object class. This method is called just before an object is garbage collected. finalize() method overrides to dispose system resources, perform clean-up activities and minimize memory leaks.

Syntax

1. **protected void** finalize() **throws** Throwable

Throw

Throwable - the Exception is raised by this method

Example 1

```
1. public class JavafinalizeExample1 {
2.     public static void main(String[] args)
3.    {
4.        JavafinalizeExample1 obj = new JavafinalizeExample1();
5.        System.out.println(obj.hashCode());
6.        obj = null;
```

```
7.        // calling garbage collector
8.        System.gc();
9.        System.out.println("end of garbage collection");
10.
11.   }
12.   @Override
13.   protected void finalize()
14.   {
15.       System.out.println("finalize method called");
16.   }
17. }
```

18. **Output:**

```
19.2018699554
20.end of garbage collection
21.finalize method called
```

Abstract Classes in Java

In C++, if a class has at least one pure virtual function, then the class becomes abstract. Unlike C++, in Java, a separate keyword *abstract* is used to make a class abstract.

```
// An example abstract class in Java
abstract class Shape {
    int color;

    // An abstract function (like a pure virtual function in C++)
    abstract void draw();
}
```

Following are some important observations about abstract classes in Java.

1) Like C++, in Java, an instance of an abstract class cannot be created, we can have references of abstract class type though.

```
abstract class Base {
    abstract void fun();
}
class Derived extends Base {
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {

        // Uncommenting the following line will cause compiler error as the
        // line tries to create an instance of abstract class.
        // Base b = new Base();

        // We can have references of Base type.
        Base b = new Derived();
        b.fun();
    }
}
```

Output:

```
Derived fun() called
```

2) Like C++, an abstract class can contain constructors in Java. And a constructor of abstract class is called when an instance of a inherited class is created. For example, the following is a valid Java program.

```
// An abstract class with constructor
abstract class Base {
    Base() { System.out.println("Base Constructor Called"); }
    abstract void fun();
}
class Derived extends Base {
    Derived() { System.out.println("Derived Constructor Called"); }
    void fun() { System.out.println("Derived fun() called"); }
}
class Main {
    public static void main(String args[]) {
       Derived d = new Derived();
    }
}
```

Output:

```
Base Constructor Called
Derived Constructor Called
```

3) In Java, we can have an abstract class without any abstract method. This allows us to create classes that cannot be instantiated, but can only be inherited.

```
// An abstract class without any abstract method
abstract class Base {
    void fun() { System.out.println("Base fun() called"); }
}

class Derived extends Base { }

class Main {
    public static void main(String args[]) {
        Derived d = new Derived();
        d.fun();
    }
}
```

Output:

```
Base fun() called
```

4) Abstract classes can also have final methods (methods that cannot be overridden). For example, the following program compiles and runs fine.

```
// An abstract class with a final method
abstract class Base {
    final void fun() { System.out.println("Derived fun() called"); }
}
```

```
class Derived extends Base {}

class Main {
    public static void main(String args[]) {
       Base b = new Derived();
       b.fun();
    }
}
```

Output:

```
Derived fun() called
```

Abstract Methods in Java with Examples

Sometimes, we require just method declaration in super-classes. This can be achieve by specifying the **abstract** type modifier. These methods are sometimes referred to as *subclasser responsibility* because they have no implementation specified in the super-class. Thus, a subclass must override them to provide method definition. To declare an abstract method, use this general form:

```
abstract type method-name(parameter-list);
```

As you can see, no method body is present. Any concrete class(i.e. class without abstract keyword) that extends an abstract class must override all the abstract methods of the class.

Important rules for abstract methods:

- Any class that contains one or more abstract methods must also be declared abstract
- The following are various **illegal combinations** of other modifiers for methods with respect to *abstract* modifier:
  1. final
  2. abstract native
  3. abstract synchronized
  4. abstract static
  5. abstract private
  6. abstract strictfp

Consider the following Java program, that illustrate the use of *abstract* keyword with classes and methods.

```
// A java program to demonstrate

// use of abstract keyword.



// abstract class

abstract class A {

    // abstract method

    // it has no body

    abstract void m1();
```

```
    // concrete methods are still
    // allowed in abstract classes
    void m2()
    {
        System.out.println("This is "
                           + "a concrete method.");
    }
}

// concrete class B
class B extends A {
    // class B must override m1() method
    // otherwise, compile-time
    // exception will be thrown
    void m1()
    {
        System.out.println("B's "
                           + "implementation of m2.");
    }
}

// Driver class
public class AbstractDemo {
    public static void main(String args[])
    {
        B b = new B();
        b.m1();
        b.m2();
```

```
    }
}
```

Output:

```
B's implementation of m2.
This is a concrete method.
```

Note: Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of super-class references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

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.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

Understanding Java Access Modifiers

Let's understand the 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

2. PACKEGES: PUTTING CLASSES TOGETHER:

Introduction:

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 form, 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.

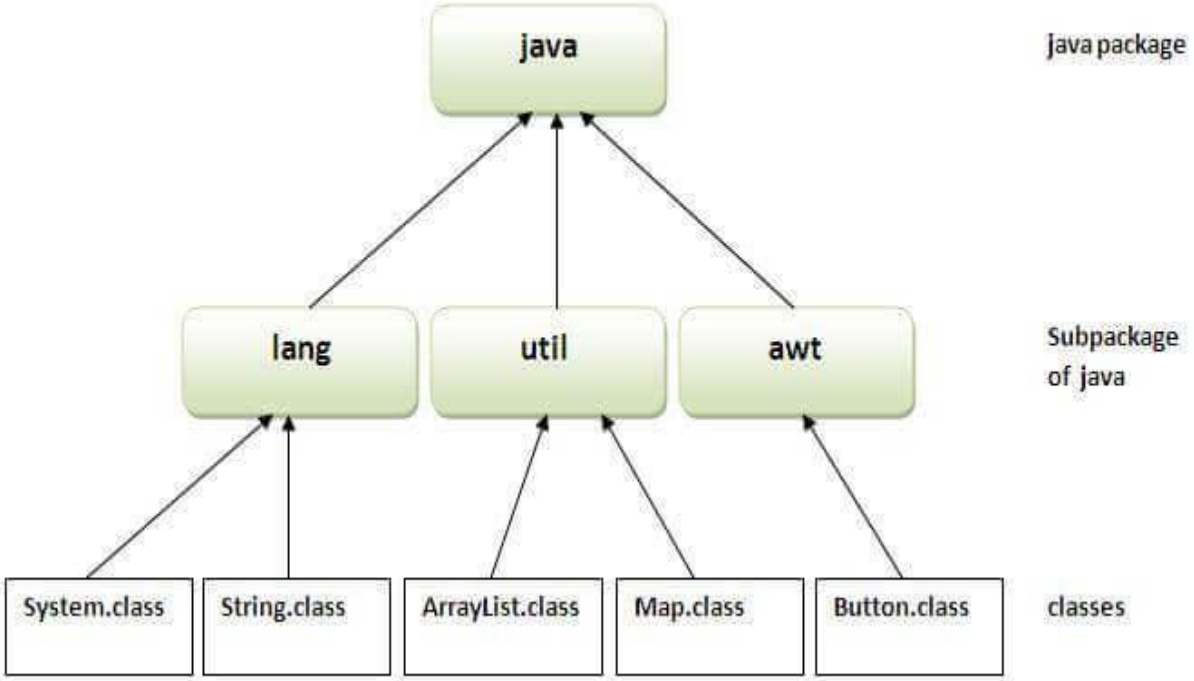
Here, we will have the detailed learning of creating and using user-defined packages.

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 collision.



Java API Packages

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

It is generally used when two packages have 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 interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.

Note: Sequence of the program must be package then import then class.

How packages work?

Package names and directory structure are closely related. For example if a package name is *college.staff.cse*, then there are three directories, *college*, *staff* and *cse* such that *cse* is present in *staff* and *staff* is present *college*. Also, the directory *college* is accessible through CLASSPATH variable, i.e., path of parent directory of college is present in CLASSPATH. The idea is to make sure that classes are easy to locate.
Package naming conventions : Packages are named in reverse order of domain names, i.e., org.CodeForIT.practice. For example, in a college, the recommended convention is college.tech.cse, college.tech.ee, college.art.history, etc.
Adding a class to a Package : We can add more classes to a created package by using package name at the top of the program and saving it in the package directory. We need a new **java** file to define a public class, otherwise we can add the new class to an existing **.java** file and recompile it.
Subpackages: Packages that are inside another package are the **subpackages**. These are not imported by default, they have to imported explicitly. Also, members of a subpackage have no access privileges, i.e., they are considered as different package for protected and default access specifiers.
Example :

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

util is a subpackage created inside **java** package.

Accessing classes inside a package

Consider following two statements :

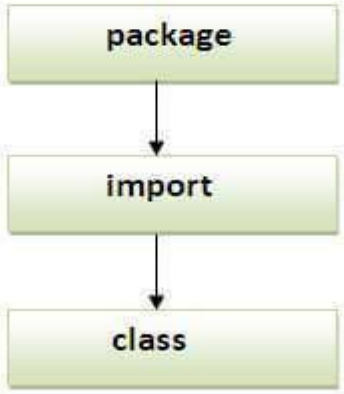
```
// import the Vector class from util package.
import java.util.vector;


// import all the classes from util package
import java.util.*;
```

- First Statement is used to import **Vector** class from **util** package which is contained inside **java**.
- Second statement imports all the classes from **util** package.

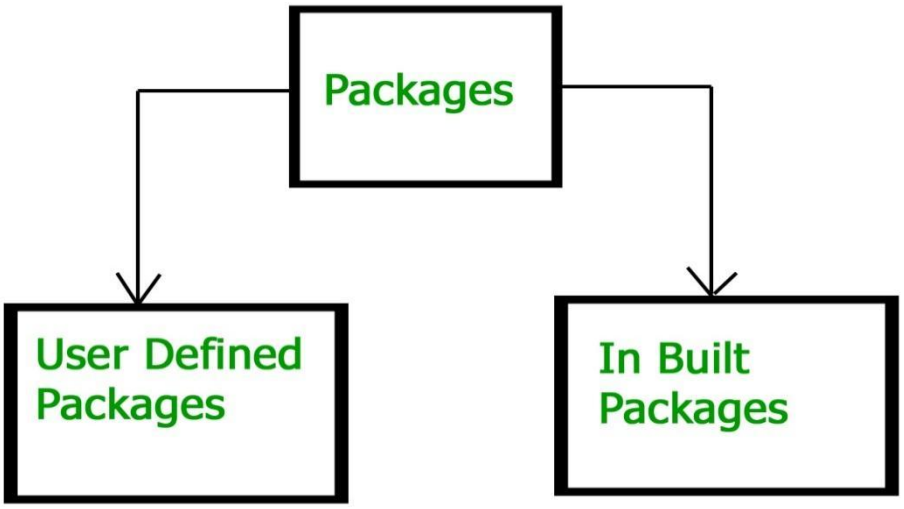
```
// All the classes and interfaces of this package
// will be accessible but not subpackages.
import package.*;


// Only mentioned class of this package will be accessible.
```

```
import package.classname;


// Class name is generally used when two packages have the same
// class name. For example in below code both packages have
// date class so using a fully qualified name to avoid conflict
import java.util.Date;
import my.packag.Date;
```

```
// Java program to demonstrate accessing of members when
// corresponding classes are imported and not imported.
import java.util.Vector;

public class ImportDemo
{
   public ImportDemo()
   {
      // java.util.Vector is imported, hence we are
      // able to access directly in our code.
      Vector newVector = new Vector();

      // java.util.ArrayList is not imported, hence
      // we were referring to it using the complete
      // package.
      java.util.ArrayList newList = new java.util.ArrayList();
   }

   public static void main(String arg[])
   {
      new ImportDemo();
   }
}
```



Types of packages:



Built-in Packages(System Packages)
These packages consist of a large number of classes which are a part of Java **API**.Some of the commonly used built-in packages are:
1) **java.lang:** Contains language support classes(e.g classed which defines primitive data types, math operations). This package is automatically imported.
2) **java.io:** Contains classed for supporting input / output operations.
3) **java.util:** Contains utility classes which implement data structures like Linked List, Dictionary and support ; for Date / Time operations.
4) **java.applet:** Contains classes for creating Applets.
5) **java.awt:** Contain classes for implementing the components for graphical user interfaces (like button , ;menus etc).
6) **java.net:** Contain classes for supporting networking operations.

User-defined packages
These are the packages that are defined by the user. First we create a directory **myPackage** (name should be same as the name of the package). Then create the **MyClass** inside the directory with the first statement being the **package names**.

```
// Name of the package must be same as the directory
```

```
// under which this file is saved
package myPackage;

public class MyClass
{
    public void getNames(String s)
    {
        System.out.println(s);
    }
}
```

Now we can use the **MyClass** class in our program.

```
/* import 'MyClass' class from 'names' myPackage */
import myPackage.MyClass;

public class PrintName
{
   public static void main(String args[])
   {
      // Initializing the String variable
      // with a value
      String name = "CodeForIT";

      // Creating an instance of class MyClass in
      // the package.
      MyClass obj = new MyClass();

      obj.getNames(name);
   }
}
```

Note : MyClass.java must be saved inside the **myPackage** directory since it is a part of the package.

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 definded 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 package is domain.company.package e.g. com.javatpoint.bean or org.sssit.dao.

Example of Subpackage

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

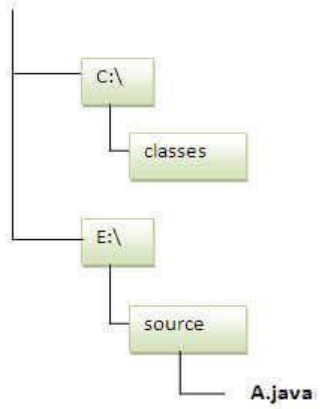
To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

```
Output:Hello subpackage
```

How to send the class file to another directory or drive?

There is a scenario, I want to put the class file of A.java source file in classes folder of c: drive. For example:




```
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. }
```

To Compile:

e:\sources> javac -d c:\classes Simple.java

To Run:

To run this program from e:\source directory, you need to set classpath of the directory where the class file resides.

e:\sources> set classpath=c:\classes;.;

e:\sources> java mypack.Simple

Another way to run this program by -classpath switch of java:

The -classpath switch can be used with javac and java tool.

To run this program from e:\source directory, you can use -classpath switch of java that tells where to look for class file. For example:

e:\sources> java -classpath c:\classes mypack.Simple

```
Output:Welcome to package
```

Ways to load the class files or jar files

There are two ways to load the class files temporary and permanent.

- Temporary
  - By setting the classpath in the command prompt

- By -classpath switch
- Permanent
  - By setting the classpath in the environment variables
  - By creating the jar file, that contains all the class files, and copying the jar file in the jre/lib/ext folder.

Rule: There can be only one public class in a java source file and it must be saved by the public class name.

```
1. //save as C.java otherwise Compilte Time Error
2.
3. class A{}
4. class B{}
5. public class C{}
```

How to put two public classes in a package?

If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:

```
1. //save as A.java
2.
3. package javatpoint;
4. public class A{}
1. //save as B.java
2. package javatpoint;
3. public class B{}
```

What is static import feature of Java5?

Click Static Import feature of Java5.