Classes, Objects and Methods

Content

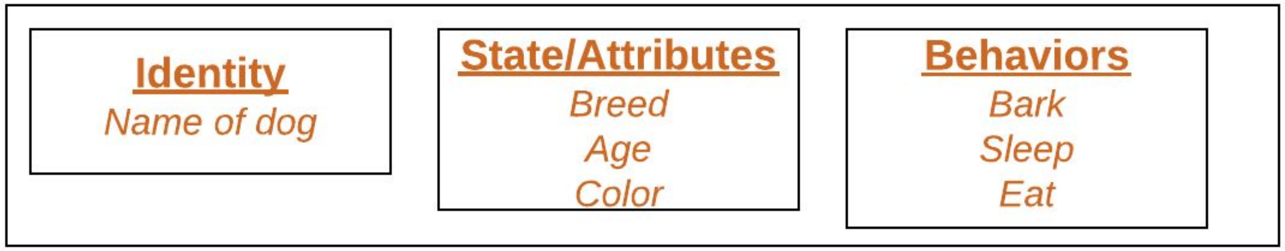
- Class
- Object
- Constructor & Constructor Overloading
- Method Overloading
- Recursion, Passing and Returning object form Method
- new operator
- this and static keyword
- finalize() method
- Access control & modifiers
- Nested class, Inner class, Anonymous inner class, Abstract class

Java Class

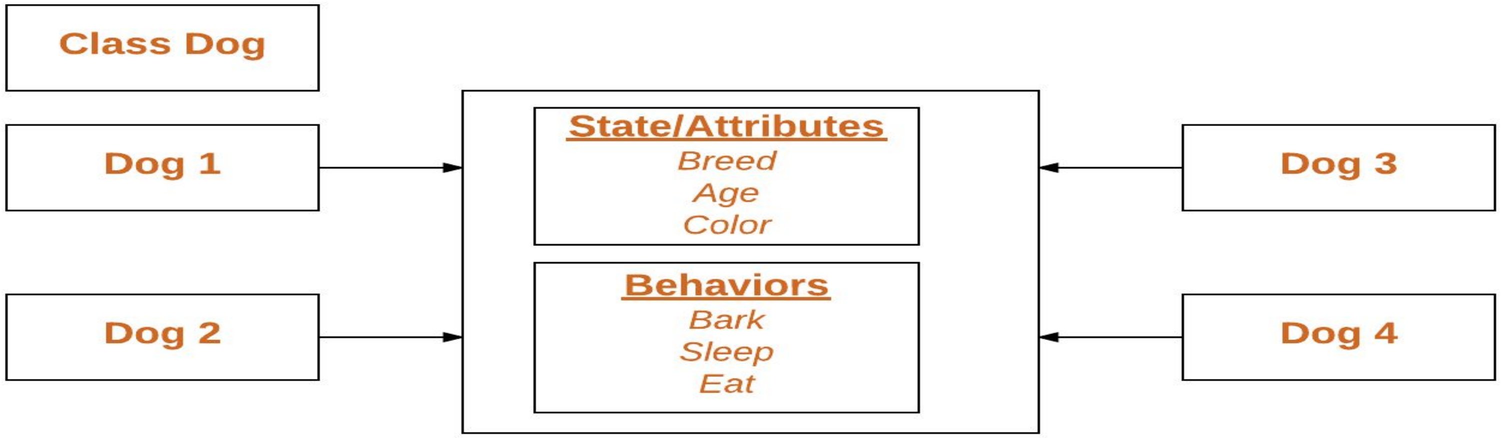
- A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:
- **Modifiers**: A class can be public or has default access
- **class keyword:** class keyword is used to create a class.
- **Class name:** The name should begin with an initial letter (capitalized by convention).
- **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.
- **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.
- **Body:** The class body surrounded by braces, { }.

Object

- It is a basic unit of Object-Oriented Programming and represents the real life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :
- **State**: It is represented by attributes of an object. It also reflects the properties of an object.
- **Behaviour**: It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity**: It gives a unique name to an object and enables one object to interact with other objects.



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.



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

-

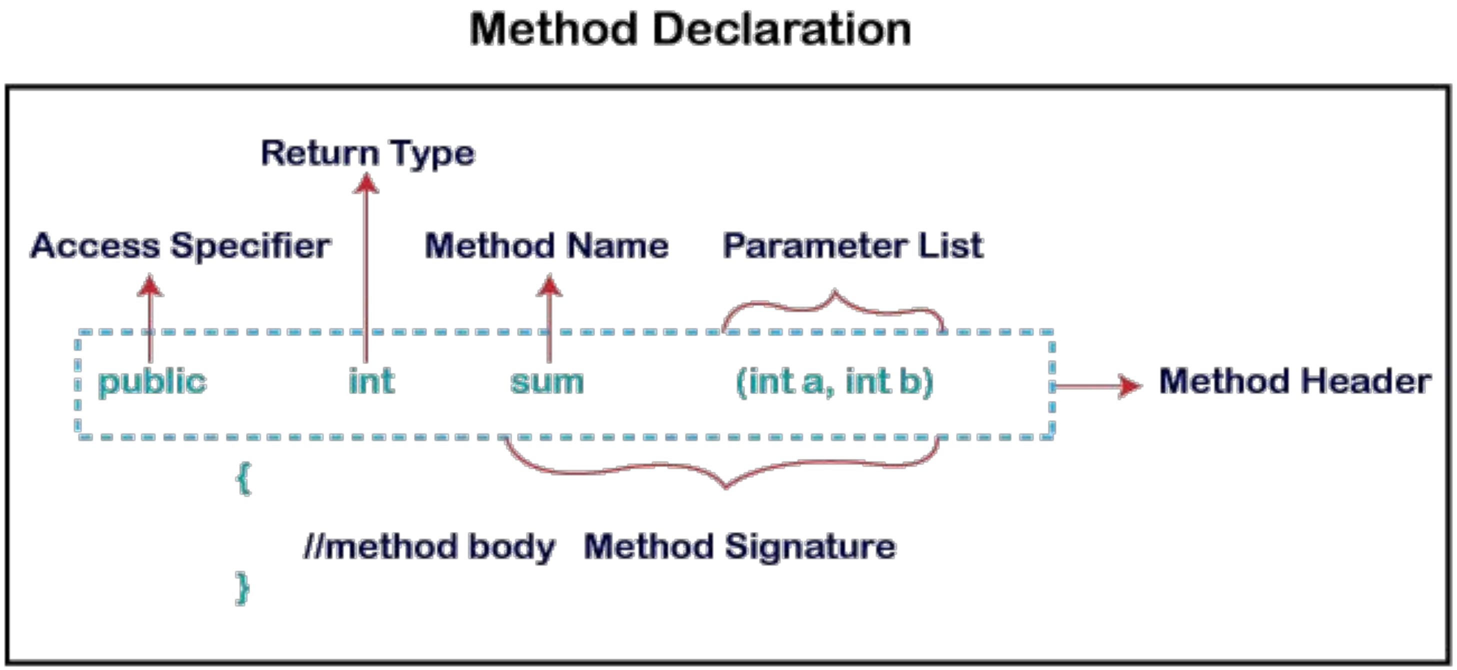
```
class Student{

 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
Student s2 = new Student();l
   }
}
```

Method in Java

- A **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **methods**
- A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code.
- We write a method once and use it many times. We do not require to write code again and again. It also provides the **easy modification** and **readability** of code, just by adding or removing a chunk of code. The method is executed only when we call or invoke it.
- The most important method in Java is the **main()** method.

Method Declaration

The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header**



- **Method Signature:** Every method has a method signature. It is a part of the method declaration. It includes the **method name** and **parameter list**.
- **Access Specifier:** Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides **four** types of access specifier:
  - **Public:** The method is accessible by all classes when we use public specifier in our application.
  - **Private:** When we use a private access specifier, the method is accessible only in the classes in which it is defined.
  - **Protected:** When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
  - **Default:** When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only

- **Return Type:** Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.
- **Method Name:** It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be **subtraction().** A method is invoked by its name.
- **Parameter List:** It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.
- **Method Body:** It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

Naming a Method

- While defining a method, remember that the method name must be a **verb** and start with a **lowercase** letter. If the method name has more than two words, the first name must be a verb followed by adjective or noun. In the multi-word method name, the first letter of each word must be in **uppercase** except the first word. For example:
  - **Single-word method name:** sum(), area()
  - **Multi-word method name:** areaOfCircle(), stringComparision()
- It is also possible that a method has the same name as another method name in the same class, it is known as **method overloading**.

Types of Method

- There are two types of methods in Java:
  - Predefined Method
  - User-defined Method
- **Predefined Method**
- In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point.
- Some pre-defined methods are **length(), equals(), compareTo(), sqrt(),** etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.
- Each and every predefined method is defined inside a class. Such as **print()** method is defined in the **java.io.PrintStream** class. It prints the statement that we write inside the method. For example, **print("Java")**, it prints Java on the console.

- **User-defined Method**
- The method written by the user or programmer is known as **a user-defined** method. These methods are modified according to the requirement.

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

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.

- Rules for creating Java constructor
  - There are two rules defined for the constructor.
  - Constructor name must be the same as its class name
  - A Constructor must have no explicit return type
  - A Java constructor cannot be abstract, static, final, and synchronized

Java Default Constructor

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

```
class Bike1{
//creating a default constructor
Bike1(){System.out.println("Bike is created");}
//main method
public static void main(String args[]){
//calling a default constructor
Bike1 b=new Bike1();
}
}
//o.p: Bike is created.
```

- **Java 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

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


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

- **Constructor Overloading in Java**
- 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.

```
class Student5{
  int id;
  String name;
  int age;
  //creating two arg constructor
  Student5(int i,String n){
  id = i;
  name = n;
  }
  //creating three arg constructor
  Student5(int i,String n,int a){
  id = i;
  name = n;
  age=a;
  }
void display(){System.out.println(id+" "+nam
e+" "+age);}

  public static void main(String args[]){
  Student5 s1 = new Student5(111,"Karan");

  Student5 s2 = new Student5(222,"Aryan",
25);
  s1.display();
  s2.display();
 }
}
```

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

Method Overloading in Java

- If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
- If we have to perform only one operation, having same name of the methods increases the readability of the program
- Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and a(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.
- Different ways to overload the method
- There are two ways to overload the method in java
  - By changing number of arguments
  - By changing the data type

- Method Overloading: changing no. of arguments
- In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.
- In this example, we are creating static methods so that we don't need to create instance for calling methods.

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

- 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.

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

- **Recursion in Java**
- Recursion in java is a process in which a method calls itself continuously. A method in java that calls itself is called recursive method.
- It makes the code compact but complex to understand.
- Syntax:

```
returntype methodname(){
//code to be executed
methodname();//calling same method
}
```

```
public class RecursionExample1 {
static void p(){
System.out.println("hello");
p();
}

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

```
public class RecursionExample3 {
    static int factorial(int n){
          if (n == 1)
            return 1;
          else
            return(n * factorial(n-1));
    }

public static void main(String[] args) {
System.out.println("Factorial of 5 is: "+factorial(5));
}
}
```

Passing and Returning Objects in Java

- Java is strictly pass by value, the precise effect differs between whether a primitive type or a reference type is passed.
- When we pass a primitive type to a method, it is passed by value. But when we pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.
- Java does this interesting thing that's sort of a hybrid between pass-by-value and pass-by-reference.
- While creating a variable of a class type, we only create a reference to an object. Thus, when we pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument.
- This effectively means that objects act as if they are passed to methods by use of call-by-reference.
- Changes to the object inside the method do reflect in the object used as an argument.

```
public class PassByValue {
  static int k =10;
  static void passPrimitive(int j) {
    System.out.println("the value of passed primitive is " + j);
    j = j + 1;
  }
  static void passReference(EmployeeTest emp) {
    EmployeeTest reference = emp;
    System.out.println("the value of name property of our object is "+ emp.getName());
    reference.setName("Bond");
  }
  public static void main(String[] args) {
    EmployeeTest ref = new EmployeeTest();
    ref.setName("James");
    passPrimitive(k);
    System.out.println("Value of primitive after get passed to method is "+ k);
    passReference(ref);
    System.out.println("Value of property of object after reference get passed to method
     is "+         ref.getName());
  }
}
```

```
class EmployeeTest {
   String name;
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
}
```

Output:

```
the value of passed primitive is 10
Value of primitive after get passed to method is 10
the value of name property of our object is James
Value of property of object after reference get passed to
    method is Bond
```

static keyword

- The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.
- The static can be:
  - Variable (also known as a class variable)
  - Method (also known as a class method)
  - Block
  - Nested class

- **Java static variable**
- If you declare any variable as static, it is known as a static variable.
- The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.
- The static variable gets memory only once in the class area at the time of class loading.
- Advantages of static variable
  - It makes your program **memory efficient** (i.e., it saves memory).
  - Understanding the problem without static variable

```
class Student{
    int rollno;
    String name;
   static String college="IITE";
}
```

- **Java static method**

- If you apply static keyword with any method, it is known as static method.
- A static method belongs to the class rather than the object of a class.
- A static method can be invoked without the need for creating an instance of a class.
- A static method can access static data member and can change the value of it.

```
class Student{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of s
     tatic variable
    static void change(){
    college = "BBDIT";
    }
    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n;
    }
    //method to display values
    void display(){System.out.println(rollno
     +" "+name+" "+college);}
}
```

```
//Test class to create and display the values of
object
public class TestStaticMethod{
   public static void main(String args[]){
   Student.change();//calling change method
   //creating objects
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan");
   Student s3 = new Student(333,"Sonoo");
   //calling display method
   s1.display();
   s2.display();
   s3.display();
   }
}
```

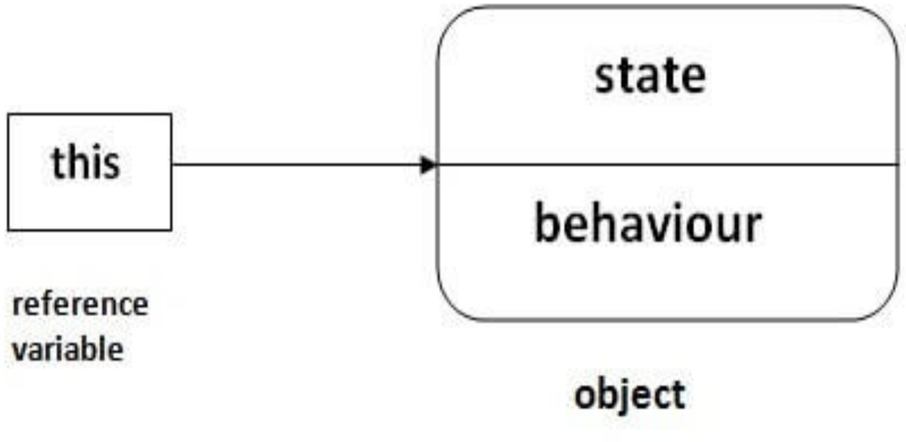
```
111 Karan BBDIT
```

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

```
class A2{
  static{ System.out.println("static block is invoked");}

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

this keyword in java

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



- Usage of java this keyword
  - this can be used to refer current class instance variable.
  - this can be used to invoke current class method (implicitly)
  - this() can be used to invoke current class constructor.
  - this can be passed as an argument in the method call.
  - this can be passed as argument in the constructor call.
  - this can be used to return the current class instance from the method.

- this: to refer current class instance variable
- The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.
- Understanding the problem without this keyword

```
//Student.java
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
rollno=rollno;
name=name;
fee=fee;
}
void display(){System.out.println(rollno+" "+na
me+" "+fee);}
}
class TestThis1{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);  /
/ 0 null 0.0
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

```
class Student{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee){
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class TestThis2{
public static void main(String args[]){
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}}
```

- this: to invoke current class method
- You may invoke the method of the current class by using the this keyword. If you don't use the this keyword, compiler automatically adds this keyword while invoking the method.

```
class A{
void m(){System.out.println("hello m");}
void n(){
System.out.println("hello n");
//m();     //same as this.m()
this.m();
}
}
class TestThis4{
public static void main(String args[]){
A a=new A();
a.n();
}}
```

- this() : to invoke current class constructor
- The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.
- **Calling default constructor from parameterized constructor:**

```
class A{
A(){System.out.println("hello a");}
A(int x){
this();
System.out.println(x);
}
}
class TestThis5{
public static void main(String args[]){
A a=new A(10);
}}
```

- this: to pass as an argument in the method
- The this keyword can also be passed as an argument in the method. It is mainly used in the event handling. Let's see the example:

```
class S2{
 void m(S2 obj){
 System.out.println("method is invoked");
 }
 void p(){
 m(this);
 }
 public static void main(String args[]){
 S2 s1 = new S2();
 s1.p();
 }
}
```

- Application of this that can be passed as an argument:
  - In event handling (or) in a situation where we have to provide reference of a class to another one. It is used to reuse one object in many methods.

- this: to pass as argument in the constructor call
- We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes

```
class B{
 A4 obj;
 B(A4 obj){
  this.obj=obj;
 }
 void display(){
  System.out.println(obj.data);//using data member of A4 class
 }
}

class A4{
 int data=10;
 A4(){
  B b=new B(this);
  b.display();
 }
 public static void main(String args[]){
  A4 a=new A4();
 }
}
```

- this keyword can be used to return current class instance
- We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive).

```
class A{
 getA(){
return this;
}
void msg(){System.out.println("Hello java");}
}
class Test1{
public static void main(String args[]){
new A().getA().msg();
}
}
```

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
- **protected void** finalize() **throws** Throwable

```
public class JavafinalizeExample1 {
     public static void main(String[] args)
    {
        JavafinalizeExample1 obj = new JavafinalizeExample1();
        System.out.println(obj.hashCode());
        obj = null;
        // calling garbage collector
        System.gc();
        System.out.println("end of garbage collection");

    }
    @Override
    protected void finalize()
    {
        System.out.println("finalize method called");
    }
}
```

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

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

- **Private**
- The private access modifier is accessible only within the class.
- **Simple example of private access modifier**
  - In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

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

Example of default access modifier

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

```
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
```

```
package pack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

- **Protected**
- The **protected access modifier** is accessible within package and outside the package but through inheritance only.
- The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.
- It provides more accessibility than the default modifer.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hell
o");}
}
//save by B.java
package mypack;
import pack.*;

class B {
 public static void main(String args[]){
  B obj = new B();
  obj.msg();  //Compile time error
 }
}
```

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

```
//save by A.java

package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
//save by B.java

package mypack;
import pack.*;

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

Java Inner Classes

- **Java inner class** or nested class is a class which is declared inside the class or interface.
- We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.
- Additionally, it can access all the members of outer class including private data members and methods.
- Syntax of Inner class

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

- Advantage of java inner classes
- Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.
- Nested classes are used **to develop more readable and maintainable code** because it logically group classes and interfaces in one place only.
- **Code Optimization**: It requires less code to write.

- **Types of Nested classes**
- There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.
- Non-static nested class (inner class)
  - Member inner class
  - Anonymous inner class
  - Local inner class
- Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

Java Member inner class

- A non-static class that is created inside a class but outside a method is called member inner class.
- Syntax:

```
class Outer{
 //code
 class Inner{
  //code
 }
}
```

```
class TestMemberOuter1{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data)
;}
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOu
ter1();
  TestMemberOuter1.Inner in=obj.new Inner();

  in.msg();
 }
}
```

- **Java Anonymous inner class**
- A class that have no name is known as anonymous inner class in java. It should be used if you have to override method of class or interface. Java Anonymous inner class can be created by two ways:
- Class (may be abstract or concrete).
- Interface

Java anonymous inner class example using class

```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[
]){
  Person p=new Person(){
  void eat(){System.out.println("nic
e fruits");}
  };
  p.eat();
 }
}
```

- **Java Local inner class**
- A class i.e. created inside a method is called local inner class in java. If you want to invoke the methods of local inner class, you must instantiate this class inside the method.

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

- **Java static nested class**
- A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.
- It can access static data members of outer class including private.
- Static nested class cannot access non-static (instance) data member or method.

```
class TestOuter1{
 static int data=30;

 static class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
 TestOuter1.Inner obj=new TestOuter1.Inner();
 obj.msg();
 }
}
```

- **Java Nested Interface**
- An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.
- Points to remember for nested interfaces
- There are given some points that should be remembered by the java programmer.
- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitely.
- Syntax of nested interface which is declared within the interface

```
interface interface_name{
 ...
 interface nested_interface_name{
  ...
 }
}
```

```
interface Showable{
  void show();
  interface Message{
   void msg();
  }
}
class TestNestedInterface1 implements Showable.Message{
 public void msg(){System.out.println("Hello nested interface");}

 public static void main(String args[]){
  Showable.Message message=new TestNestedInterface1();//upcasti
     ng here
  message.msg();
 }
}
```