Constructer:

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a method. However, constructors have no explicit return type.

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other start-up procedures required to create a fully formed object.

All classes have constructors, whether you define one or not, because Java automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Syntax

Following is the syntax of a constructor −

class ClassName {

ClassName() {

}

}

Java allows two types of constructors namely −

* No argument Constructors
* Parameterized Constructors

No argument Constructors

As the name specifies the no argument constructors of Java does not accept any parameters instead, using these constructors the instance variables of a method will be initialized with fixed values for all objects.

Example

Public class MyClass {

Int num;

MyClass() {

num = 100;

}

}

You would call constructor to initialize objects as follows

public class ConsDemo {

public static void main(String args[]) {

MyClass t1 = new MyClass();

MyClass t2 = new MyClass();

System.out.println(t1.num + " " + t2.num);

}

}

This would produce the following result

100 100

Parameterized Constructors

Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

Example

Here is a simple example that uses a constructor −

// A simple constructor.

class MyClass {

int x;

// Following is the constructor

MyClass(int i ) {

x = i;

}

}

You would call constructor to initialize objects as follows −

public class ConsDemo {

public static void main(String args[]) {

MyClass t1 = new MyClass( 10 );

MyClass t2 = new MyClass( 20 );

System.out.println(t1.x + " " + t2.x);

}

}

This would produce the following result −

10 20

Overloading:

In the previous chapter, we talked about superclasses and subclasses. If a class inherits a method from its superclass, then there is a chance to override the method provided that it is not marked final.

The benefit of overriding is: ability to define a behavior that's specific to the subclass type, which means a subclass can implement a parent class method based on its requirement.

In object-oriented terms, overriding means to override the functionality of an existing method.

### **Example**

Let us look at an example.

class Animal {

public void move() {

System.out.println("Animals can move");

}

}

class Dog extends Animal {

public void move() {

System.out.println("Dogs can walk and run");

}

}

public class TestDog {

public static void main(String args[]) {

Animal a = new Animal(); // Animal reference and object

Animal b = new Dog(); // Animal reference but Dog object

a.move(); // runs the method in Animal class

b.move(); // runs the method in Dog class

}

}

This will produce the following result −

### **Output**

Animals can move

Dogs can walk and run

In the above example, you can see that even though **b** is a type of Animal it runs the move method in the Dog class. The reason for this is: In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.

Therefore, in the above example, the program will compile properly since Animal class has the method move. Then, at the runtime, it runs the method specific for that object.

Consider the following example −

### **Example**

[Live Demo](http://tpcg.io/VHj8iU)

class Animal {

public void move() {

System.out.println("Animals can move");

}

}

class Dog extends Animal {

public void move() {

System.out.println("Dogs can walk and run");

}

public void bark() {

System.out.println("Dogs can bark");

}

}

public class TestDog {

public static void main(String args[]) {

Animal a = new Animal(); // Animal reference and object

Animal b = new Dog(); // Animal reference but Dog object

a.move(); // runs the method in Animal class

b.move(); // runs the method in Dog class

b.bark();

}

}

This will produce the following result −

### **Output**

TestDog.java:26: error: cannot find symbol

b.bark();

^

symbol: method bark()

location: variable b of type Animal

1 error

This program will throw a compile time error since b's reference type Animal doesn't have a method by the name of bark.

## Rules for Method Overriding

* The argument list should be exactly the same as that of the overridden method.
* The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
* The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overridding method in the sub class cannot be either private or protected.
* Instance methods can be overridden only if they are inherited by the subclass.
* A method declared final cannot be overridden.
* A method declared static cannot be overridden but can be re-declared.
* If a method cannot be inherited, then it cannot be overridden.
* A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
* A subclass in a different package can only override the non-final methods declared public or protected.
* An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
* Constructors cannot be overridden.

## Using the super Keyword

When invoking a superclass version of an overridden method the **super** keyword is used.

### **Example**

[Live Demo](http://tpcg.io/r51jFh)

class Animal {

public void move() {

System.out.println("Animals can move");

}

}

class Dog extends Animal {

public void move() {

super.move(); // invokes the super class method

System.out.println("Dogs can walk and run");

}

}

public class TestDog {

public static void main(String args[]) {

Animal b = new Dog(); // Animal reference but Dog object

b.move(); // runs the method in Dog class

}

}

This will produce the following result −

### **Output**

Animals can move

Dogs can walk and run

Garbage Collection:

**Introduction**

* In C/C++, programmer is responsible for both creation and destruction of objects. Usually programmer neglects destruction of useless objects. Due to this negligence, at certain point, for creation of new objects, sufficient memory may not be available and entire program will terminate abnormally causing **OutOfMemoryErrors**.
* But in Java, the programmer need not to care for all those objects which are no longer in use. Garbage collector destroys these objects.
* Garbage collector is best example of [Daemon thread](https://www.geeksforgeeks.org/daemon-thread-java/) as it is always running in background.
* Main objective of Garbage Collector is to free heap memory by destroying **unreachable objects**.

**Important terms :**

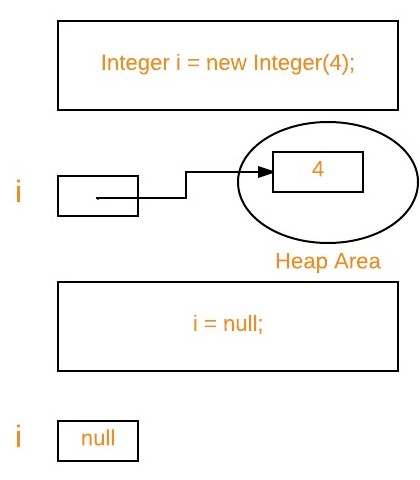
1. **Unreachable objects :**An object is said to be unreachable iff it doesn’t contain any reference to it. Also note that objects which are part of [island of isolation](https://www.geeksforgeeks.org/island-of-isolation-in-java/) are also unreachable.

Integer i = new Integer(4);

// the new Integer object is reachable via the reference in 'i'

i = null;

// the Integer object is no longer reachable.



1. **Eligibility for garbage collection :**An object is said to be eligible for GC(garbage collection) iff it is unreachable. In above image, after *i = null;* integer object 4 in heap area is eligible for garbage collection.

**Ways to make an object eligible for GC**

* Even though the programmer is not responsible to destroy useless objects but it is highly recommended to make an object unreachable(thus eligible for GC) if it is no longer required.
* There are generally four different ways to make an object eligible for garbage collection.
  + 1. Nullifying the reference variable
    2. Re-assigning the reference variable
    3. Object created inside method
    4. [Island of Isolation](https://www.geeksforgeeks.org/island-of-isolation-in-java/)

All above ways with examples are discussed in separate article : [How to make object eligible for garbage collection](https://www.geeksforgeeks.org/how-to-make-object-eligible-for-garbage-collection/)

**Ways for requesting**[**JVM**](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/)**to run Garbage Collector**

* Once we made object eligible for garbage collection, it may not destroy immediately by the garbage collector. Whenever JVM runs the Garbage Collector program, then only the object will be destroyed. But when JVM runs Garbage Collector, we can not expect.
* We can also request JVM to run Garbage Collector. There are two ways to do it :
  + 1. **Using *System.gc()* method** : System class contain static method *gc()* for requesting JVM to run Garbage Collector.
    2. **Using *Runtime.getRuntime().gc()* method** : [Runtime class](https://www.geeksforgeeks.org/java-lang-runtime-class-in-java/) allows the application to interface with the JVM in which the application is running. Hence by using its gc() method, we can request JVM to run Garbage Collector.

|  |
| --- |
| // Java program to demonstrate requesting  // JVM to run Garbage Collector  public class Test  {      public static void main(String[] args) throws InterruptedException      {          Test t1 = new Test();          Test t2 = new Test();            // Nullifying the reference variable          t1 = null;            // requesting JVM for running Garbage Collector          System.gc();            // Nullifying the reference variable          t2 = null;            // requesting JVM for running Garbage Collector          Runtime.getRuntime().gc();        }        @Override      // finalize method is called on object once      // before garbage collecting it      protected void finalize() throws Throwable      {          System.out.println("Garbage collector called");          System.out.println("Object garbage collected : " + this);      }  } |

Output:

Garbage collector called

Object garbage collected : Test@46d08f12

Garbage collector called

Object garbage collected : Test@481779b8

**Note :**

* + 1. There is no guarantee that any one of above two methods will definitely run Garbage Collector.
    2. The call *System.gc()* is effectively equivalent to the call : *Runtime.getRuntime().gc()*

**Finalization**

* Just before destroying an object, Garbage Collector calls *finalize()* method on the object to perform cleanup activities. Once *finalize()* method completes, Garbage Collector destroys that object.
* *finalize()* method is present in [Object class](https://www.geeksforgeeks.org/object-class-in-java/) with following prototype.
* protected void finalize() throws Throwable

Based on our requirement, we can override *finalize()* method for perform our cleanup activities like closing connection from database.

**Note :**

* 1. The *finalize()*method called by Garbage Collector not [JVM](https://www.geeksforgeeks.org/jvm-works-jvm-architecture/). Although Garbage Collector is one of the module of JVM.
  2. [Object class](https://www.geeksforgeeks.org/object-class-in-java/) *finalize()* method has empty implementation, thus it is recommended to override *finalize()* method to dispose of system resources or to perform other cleanup.
  3. The *finalize()*method is never invoked more than once for any given object.
  4. If an uncaught exception is thrown by the *finalize()* method, the exception is ignored and finalization of that object terminates.

For examples on *finalize()* method, please see [Output of Java programs | Set 10 (Garbage Collection)](https://www.geeksforgeeks.org/output-of-java-programs-set-10-garbage-collection/)

**Let’s take a real-life example, where we use the concept of garbage collector.**

Suppose you go for the internship at GeeksForGeeks and their you were told to write a program, to count the number of Employees working in the company(excluding interns).To make this program, you have to use the concept of a garbage collector.  
This is the actual task you were given at the company:-

**Q.**Write a program to create a class called Employee having the following data members.  
1.An ID for storing unique id allocated to every employee.  
2.Name of employee.  
3.age of an employee.

Also, provide the following methods-

1. A parameterized constructor to initialize name and age. The ID should be initialized in this constructor.
2. A method show() to display ID, name, and age.
3. A method showNextId() to display the ID of the next employee.

Now any beginner, who doesn’t have knowledge on garbage collector will code like this:

|  |
| --- |
| //Program to count number  //of employees working  //in a company    class Employee  {      private int ID;      private String name;      private int age;      private static int nextId=1;      //it is made static because it      // is keep common among all and      // shared by all objects      public Employee(String name,int age)      {          this.name = name;          this.age = age;          this.ID = nextId++;      }      public void show()      {          System.out.println          ("Id="+ID+"\nName="+name+"\nAge="+age);      }      public void showNextId()      {          System.out.println          ("Next employee id will be="+nextId);      }  }  class UseEmployee  {      public static void main(String []args)      {          Employee E=new Employee("GFG1",56);          Employee F=new Employee("GFG2",45);          Employee G=new Employee("GFG3",25);          E.show();          F.show();          G.show();          E.showNextId();          F.showNextId();          G.showNextId();                { //It is sub block to keep              // all those interns.              Employee X=new Employee("GFG4",23);              Employee Y=new Employee("GFG5",21);              X.show();              Y.show();              X.showNextId();              Y.showNextId();          }          //After countering this brace, X and Y          //will be removed.Therefore,          //now it should show nextId as 4.          E.showNextId();//Output of this line          //should be 4 but it will give 6 as output.      }  } |

Output:

Id=1

Name=GFG1

Age=56

Id=2

Name=GFG2

Age=45

Id=3

Name=GFG3

Age=25

Next employee id will be=4

Next employee id will be=4

Next employee id will be=4

Id=4

Name=GFG4

Age=23

Id=5

Name=GFG5

Age=21

Next employee id will be=6

Next employee id will be=6

Next employee id will be=6

**Now to get the correct output:**  
Now garbage collector(gc) will see 2 objects free. Now to decrement nextId,gc(garbage collector) will call method finalize() only when we programmers have override it in our class. And as mentioned previously, we have to request gc(garbage collector) and for this, we have to write the following 3 steps before closing brace of sub-block.

1. Set references to null(i.e X = Y = null;)
2. Call, System.gc();
3. Call, System.runFinalization();

Now the correct code for counting the number of employees(excluding interns)

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // Correct code to count number  // of employees excluding interns.  class Employee  {      private int ID;      private String name;      private int age;      private static int nextId=1;      //it is made static because it      // is keep common among all and      // shared by all objects      public Employee(String name,int age)      {          this.name = name;          this.age = age;          this.ID = nextId++;      }      public void show()      {          System.out.println          ("Id="+ID+"\nName="+name+"\nAge="+age);      }      public void showNextId()      {          System.out.println          ("Next employee id will be="+nextId);      }      protected void finalize()      {          --nextId;          //In this case,          //gc will call finalize()          //for 2 times for 2 objects.      }  }    // it is closing brace of Employee class  class UseEmployee  {      public static void main(String []args)      {          Employee E=new Employee("GFG1",56);          Employee F=new Employee("GFG2",45);          Employee G=new Employee("GFG3",25);          E.show();          F.show();          G.show();          E.showNextId();          F.showNextId();          G.showNextId();            {              //It is sub block to keep              // all those interns.              Employee X=new Employee("GFG4",23);              Employee Y=new Employee("GFG5",21);              X.show();              Y.show();              X.showNextId();              Y.showNextId();              X = Y = null;              System.gc();              System.runFinalization();          }      E.showNextId();      }  } |

Output:

Id=1

Name=GFG1

Age=56

Id=2

Name=GFG2

Age=45

Id=3

Name=GFG3

Age=25

Next employee id will be=4

Next employee id will be=4

Next employee id will be=4

Id=4

Name=GFG4

Age=23

Id=5

Name=GFG5

Age=21

Next employee id will be=6

Next employee id will be=6

Next employee id will be=4