Unit III: Java as Object Oriented Programming Language- Overview

Fundamentals of JAVA, Arrays: one dimensional array, multi-dimensional array, alternative array declaration statements,

String Handling: String class methods

Classes and Methods: class fundamentals, declaring objects, assigning object reference variables, adding methods to a class, returning a value, constructors, this keyword, garbage collection, finalize() method, overloading methods, argument passing, object as parameter, returning objects, access control, static, final, nested and inner classes, command line arguments, variable - length arguments.

Java Class and Objects

Java is an object-oriented programming language. The core concept of the object-oriented approach is to break complex problems into smaller objects.

An object is any entity that has a **state** and **behavior**. For example, a bicycle is an object. It has

- **States**: idle, first gear, etc
- **Behaviors**: braking, accelerating, etc.

Before we learn about objects, let's first know about classes in Java.

Java Class

- A class is a blueprint for the object. Before we create an object, we first need to define the class.
- We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.
- Since many houses can be made from the same description, we can create many objects from a class.

Create a class in Java

We can create a class in Java using the class keyword. For example,

```
class ClassName {
  // fields
  // methods
}
```

Here, fields (variables) and methods represent the **state** and **behavior** of the object respectively.

- fields are used to store data
- methods are used to perform some operations

For our bicycle object, we can create the class as

```
class Bicycle {
  // state or field
  private int gear = 5;
  // behavior or method
  public void braking() {
    System.out.println("Working of Braking");
  }
}
```

- In the above example, we have created a class named Bicycle. It contains a field named gear and a method named braking().
- Here, Bicycle is a prototype. Now, we can create any number of bicycles using the prototype. And, all the bicycles will share the fields and methods of the prototype.

Note: We have used keywords private and public. These are known as access modifiers.

Java Objects:

An object is called an instance of a class. For example, suppose Bicycle is a class then MountainBicycle, SportsBicycle, TouringBicycle, etc can be considered as objects of the class.

Creating an Object in Java

Here is how we can create an object of a class.

```
className object = new className();
// for Bicycle class
Bicycle sportsBicycle = new Bicycle();
Bicycle touringBicycle = new Bicycle();
```

- We have used the new keyword along with the constructor of the class to create an object. Constructors are similar to methods and have the same name as the class.
- For example, Bicycle () is the constructor of the Bicycle class.
- Here, sportsBicycle and touringBicycle are the names of objects. We can use them to access fields and methods of the class.
- As you can see, we have created two objects of the class. We can create multiple objects of a single class in Java.

Note: Fields and methods of a class are also called members of the class.

Access Members of a Class

We can use the name of objects along with the . operator to access members of a class.

For example,

```
class Bicycle {

  // field of class

  int gear = 5;

  // method of class

  void braking() {

    ...

  }

}

// create object

Bicycle sportsBicycle = new Bicycle();



// access field and method

sportsBicycle.gear;

sportsBicycle.braking();
```

In the above example, we have created a class named Bicycle. It includes a field named gear and a method named braking(). Notice the statement,

```
Bicycle sportsBicycle = new Bicycle();
```

Here, we have created an object of Bicycle named sportsBicycle. We then use the object to access the field and method of the class.

- **sportsBicycle.gear** - access the field gear
- **sportsBicycle.braking()** - access the method braking()

Let's see a fully working example.

Example: Java Class and Objects

```
class Lamp {
  // stores the value for light. true if light is on. false if light is off
  boolean isOn;
  // method to turn on the light
  void turnOn() {
    isOn = true;
    System.out.println("Light on? " + isOn);
  }
  // method to turnoff the light
  void turnOff() {
    isOn = false;
    System.out.println("Light on? " + isOn);
  }
}
class ABC {
  public static void main(String[] args) {
    // create objects led and halogen
    Lamp led = new Lamp();
    Lamp halogen = new Lamp();
    // turn on the light by calling method turnOn()
    led.turnOn();
    // turn off the light by calling method turnOff()
    halogen.turnOff();
  }
}
```

Output:

```
Light on? true

Light on? false
```

In the above program, we have created a class named Lamp. It contains a variable: isOn and two methods: turnOn() and turnOff().

Inside the Main class, we have created two objects: led and halogen of the Lamp class. We then used the objects to call the methods of the class.

- **led.turnOn()** - It sets the isOn variable to true and prints the output.
- **halogen.turnOff()** - It sets the isOn variable to false and prints the output.

The variable isOn defined inside the class is also called an instance variable. It is because when we create an object of the class, it is called an instance of the class. And, each instance will have its own copy of the variable.

That is, led and halogen objects will have their own copy of the isOn variable.

Example: Create objects inside the same class

Note that in the previous example, we have created objects inside another class and accessed the members from that class.

However, we can also create objects inside the same class.

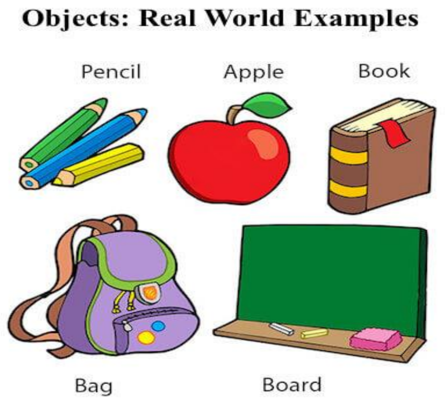
```
class Lamp {
   // stores the value for light
  // true if light is on
  // false if light is off
  boolean isOn;
  // method to turn on the light
  void turnOn() {
    isOn = true;
    System.out.println("Light on? " + isOn);
  }
  public static void main(String[] args) {
    // create an object of Lamp
    Lamp led = new Lamp();
   // access method using object
    led.turnOn();
  }
}
```

Output

```
Light on? true
```

Here, we are creating the object inside the main () method of the same class.

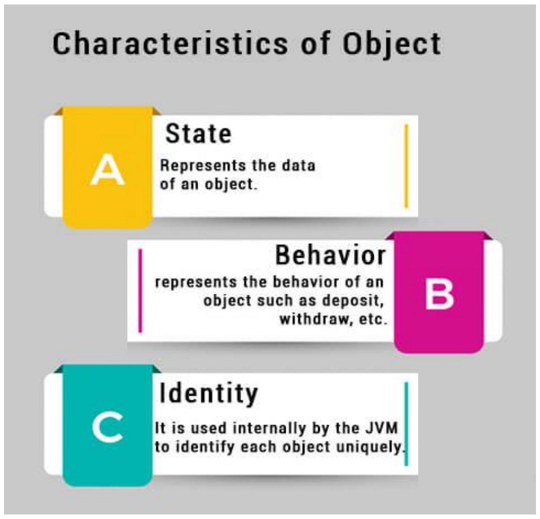
What is an object in Java



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

An object has three characteristics:

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


Characteristics of Object
A
State
Represents the data of an object.
Behavior
represents the behavior of an object such as deposit, withdraw, etc.
B
C
Identity
It is used internally by the JVM to identify each object uniquely.

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

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

Object Definitions:

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

Object and Class Example: main outside the class

- In real time development, we create classes and use it from another class. It is a better approach than previous one. Let's see a simple example, where we are having main () method in another class.
- We can have multiple classes in different Java files or single Java file. If you define multiple classes in a single Java source file, it is a good idea to save the file name with the class name which has main () method.

TestStudent1.java

```
//Java Program to demonstrate having the main method in another class
//Creating Student class.
class Student{
 int id;
 String name;      }
//Creating another class TestStudent1 which contains the main method
class TestStudent1{
 public static void main(String args[]){
  Student s1=new Student();
  System.out.println(s1.id);
  System.out.println(s1.name);
 }           }
```

Output:

```
0
Null
```

3 Ways to initialize object

There are 3 ways to initialize object in Java.

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

1) Object and Class Example: Initialization through reference

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

TestStudent2.java

```
class Student{
 int id;
 String name;
}
class TestStudent2{
 public static void main(String args[]){
  Student s1=new Student();
  s1.id=101;
  s1.name="Computer";
  System.out.println(s1.id+" "+s1.name);//printing members with a white space
 }
}
```

Output:

```
101 Computer
```

We can also create multiple objects and store information in it through reference variable.

TestStudent3.java

```
class Student{
 int id;
 String name;
}
class TestStudent3{
 public static void main(String args[]){
  //Creating objects
  Student s1=new Student();
  Student s2=new Student();
  //Initializing objects
  s1.id=101;
  s1.name="Computer";
  s2.id=102;
  s2.name="Department";
  //Printing data
  System.out.println(s1.id+" "+s1.name);
  System.out.println(s2.id+" "+s2.name);
 }
}
```

Output:

```
101 Computer
102 Department
```

2) Object and Class Example: Initialization through method

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

```
class Student{
 int rollno;
 String name;
 void insertRecord(int r, String n){
  rollno=r;
  name=n;
 }
 void displayInformation(){System.out.println(rollno+" "+name);}
}
class TestStudent4{
 public static void main(String args[]){
  Student s1=new Student();
  Student s2=new Student();
  s1.insertRecord(111,"Computer");
  s2.insertRecord(222,"Department");
  s1.displayInformation();
  s2.displayInformation();  }  }
```

Output:

```
111 Computer
222 Department
```

The object gets the memory in heap memory area. The reference variable refers to the object allocated in the heap memory area. Here, s1 and s2 both are reference variables that refer to the objects allocated in memory.

3) Object and Class Example: Initialization through a constructor:

Object and Class Example: Employee

Let's see an example where we are maintaining records of employees.

TestEmployee.java

```
class Employee{
   int id;
   String name;
   float salary;
   void insert(int i, String n, float s) {
      id=i;
      name=n;
      salary=s;
   }
   void display(){
              System.out.println(id+" "+name+" "+salary);
}    }
public class TestEmployee {
public static void main(String[] args) {
   Employee e1=new Employee();
   Employee e2=new Employee();
   Employee e3=new Employee();
   e1.insert(101,"SE",45000);
   e2.insert(102,"TE",25000);
   e3.insert(103,"BE",55000);
   e1.display();
   e2.display();
   e3.display();  }  }
```

Output:

```
101 SE 45000
102 TE 25000
103 BE 55000
```

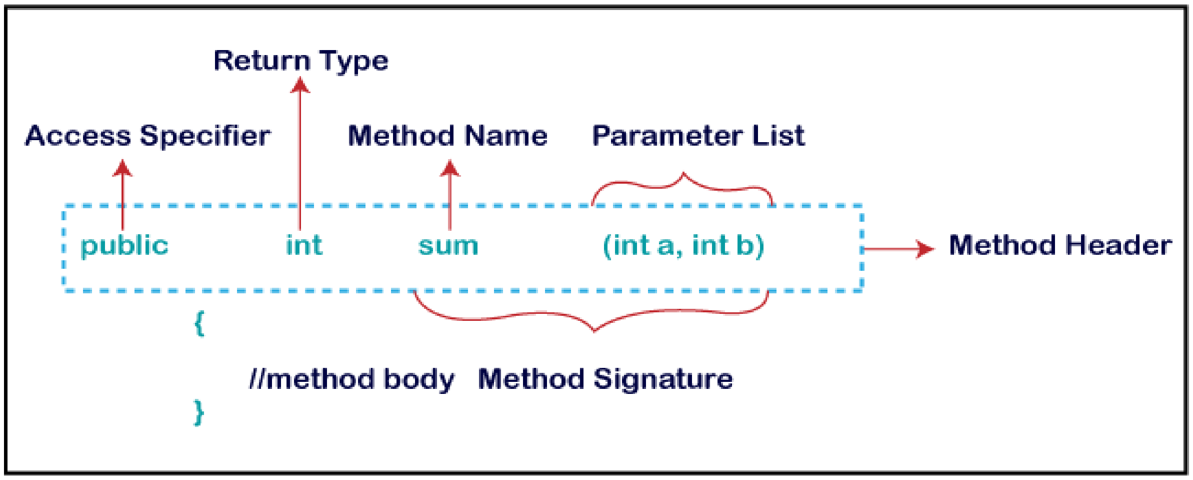
Java Methods

- A method is a block of code that performs a specific task.
- Suppose you need to create a program to create a circle and color it. You can create two methods to solve this problem:
  - a method to draw the circle
  - a method to color the circle
- Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.
- In Java, there are two types of methods:
  - **User-defined Methods**: We can create our own method based on our requirements.
  - **Standard Library Methods**: These are built-in methods in Java that are available to use.

Declaring a Java Method

The syntax to declare a method is:

Method Declaration



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:

1. **Single-word method name:** sum(), area()
2. **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

EvenOdd.java

```
import java.util.Scanner;
public class EvenOdd
{
public static void main (String args[])
{
//creating Scanner class object
Scanner scan=new Scanner(System.in);
System.out.print("Enter the number: ");
//reading value from user
int num=scan.nextInt();
//method calling
findEvenOdd(num);
}
//user defined method
public static void findEvenOdd(int num)
{
//method body
if(num%2==0)
System.out.println(num+" is even");
else
System.out.println(num+" is odd");
}
}
```

Output 1:

```
Enter the number: 12
12 is even
```

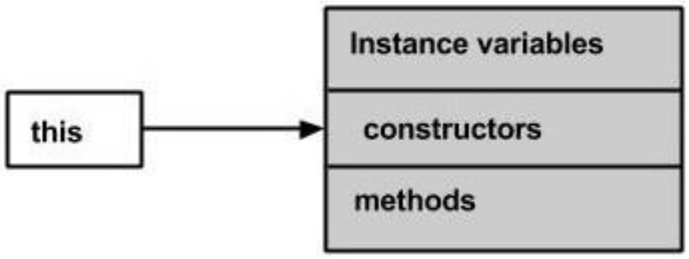
Output 2:

```
Enter the number: 99
99 is odd
```

The this Keyword inside Java Methods

- **this** is a keyword in Java which is used as a reference to the object of the current class, with in an instance method or a constructor.
- Using this you can refer the members of a class such as constructors, variables and methods.

Note − the keyword **this** is used only within instance methods or constructors.



In general, the keyword this is used to −

Differentiate the instance variables from local variables if they have same names, within a constructor or a method.

```
class Student {
   int age;
   Student(int age) {
      this.age = age;
   }
}
```

Call one type of constructor (parameterized constructor or default) from other in a class. It is known as explicit constructor invocation.

```
class Student {
   int age
   Student() {
      this(20);
   }
      Student(int age) {
      this.age = age;
   }
}
```

Example: Use of this keyword in Java Methods

Here is an example that uses *this* keyword to access the members of a class. Copy and paste the following program in a file with the name,

This_Example.java.

```
public class This_Example {
   // Instance variable num
   int num = 10;

   This_Example() {
      System.out.println("This is an example program on keyword this");
   }

   This_Example(int num) {
      // Invoking the default constructor
      this();

      // Assigning the local variable num to the instance variable num
      this.num = num;
   }

   public void greet() {
      System.out.println("Hi Welcome to This Example Program");
   }

   public void print() {
      // Local variable num
      int num = 20;

      // Printing the local variable
      System.out.println("Value of local variable num is : "+num);

      // Printing the instance variable
      System.out.println("Value of instance variable num is : "+this.num);

      // Invoking the greet method of a class
      this.greet();
   }
   public static void main(String[] args) {
      // Instantiating the class
      This_Example obj1 = new This_Example();

      // Invoking the print method
      obj1.print();

      // Passing a new value to the num variable through parameterized constructor
      This_Example obj2 = new This_Example(30);

      // Invoking the print method again
      obj2.print();
   }
}
```

Output

```
This is an example program on keyword this
Value of local variable num is : 20
Value of instance variable num is : 10
Hi Welcome to This Example Program
This is an example program on keyword this
Value of local variable num is : 20
Value of instance variable num is : 30
Hi Welcome to This Example Program
```

Java garbage collection: What is it and how does it work?

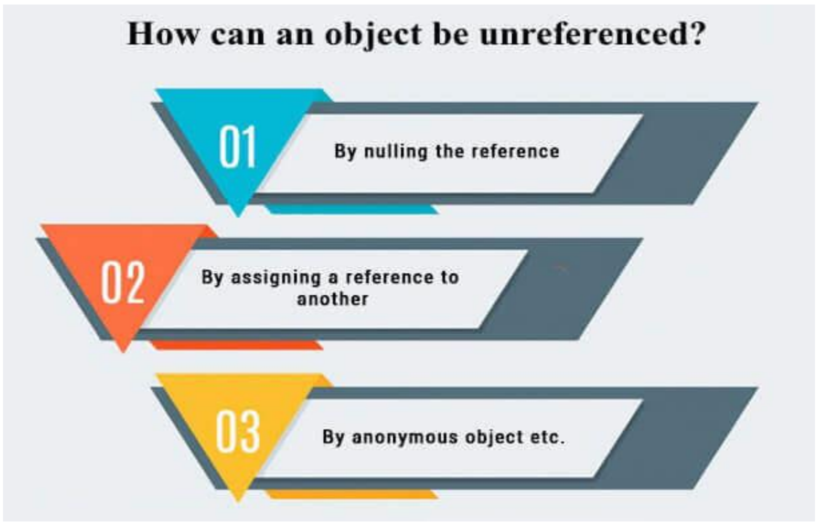
- Garbage collection in Java is the automated process of deleting code that's no longer needed or used.
- This automatically frees up memory space and ideally makes coding Java apps easier for developers.
- Java applications are compiled into bytecode that may be executed by a JVM.
- Objects are produced on the heap (the memory space used for dynamic allocation), which are then monitored and tracked by garbage collection operations.
- Most objects used in Java code are short-lived and can be reclaimed shortly after they are created.
- The garbage collector uses a mark-and-sweep algorithm to mark all unreachable objects as garbage collection, then scans through live objects to find objects that are still reachable.
- Automatic garbage collection means you don't have control over whether and when objects are deleted.
- This is in contrast to languages like C and C++, where garbage collection is handled manually. However, automatic garbage collection is popular for good reason—manual memory management is cumbersome and slows down the pace of application development.

How does garbage collection work in Java?

- During the garbage collection process, the collector scans different parts of the heap, looking for objects that are no longer in use.
- If an object no longer has any references to it from elsewhere in the application, the collector removes the object, freeing up memory in the heap.
- This process continues until all unused objects are successfully reclaimed.
- Sometimes, a developer will inadvertently write code that continues to be referenced even though it's no longer being used.
- The garbage collector will not remove objects that are being referenced in this way, leading to memory leaks.
- After memory leaks are created, it can be hard to detect the cause, so it's important to prevent memory leaks by ensuring that there are no references to unused objects.
- To ensure that garbage collectors work efficiently, the JVM separates the heap into separate spaces, and then garbage collectors use a mark-and-sweep algorithm to traverse these spaces and clear out unused objects.

How can an object be unreferenced?

There are many ways:


How can an object be unreferenced?
01
By nulling the reference
02
By assigning a reference to another
03
By anonymous object etc.

1) By nulling a reference:

1. Employee e=**new** Employee();
2. e=**null**;

2) By assigning a reference to another:

1. Employee e1=**new** Employee();
2. Employee e2=**new** Employee();
3. e1=e2;//now the first object referred by e1 is available for garbage collection

3) By anonymous object:

1. **new** Employee();

---

finalize () method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

1. **protected void** finalize(){}

Note: The Garbage collector of JVM collects only those objects that are created by new keyword. So if you have created any object without new, you can use finalize method to perform cleanup processing (destroying remaining objects).

gc() method

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

1. **public static void** gc(){}

Note: Garbage collection is performed by a daemon thread called Garbage Collector (GC). This thread calls the finalize () method before object is garbage collected.

Simple Example of garbage collection in java:

```
public class TestGarbage1{
 public void finalize(){  System.out.println("Object is garbage collected");   }
 public static void main(String args[]){
  TestGarbage1 s1=new TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 }
}
```

Output:

```
Object is garbage collected
Object is garbage collected
```

When to Use finalize() Method in Java?

Finalize () Method in Java might be used in the following scenarios:

- **Releasing system-level resources:**
  Suppose your object uses any system-level resources, you can use **finalize()** Method to free up these resources before the garbage collector reclaims your object.

- **Releasing external resources:**
  If your object has an external resource such as file handles or database connections, you can use the **finalize()** Method to release these resources before your object can be garbage collected.

- **Implementing custom clean-up procedures:**

You can use **finalize ()** Method to provide a custom clean-up procedure to your object

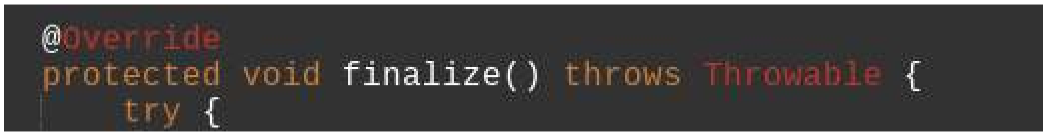
Why finalize() Method is used?

You can use **finalize()** method for the following reasons:

- You can do clean-up operations on your object before the garbage collector reclaims it
- It protects the overlooked resources used by your object and ensures its release before garbage collection.
- It helps in profiling and debugging by letting you check when the garbage collector is reclaiming your objects.

Overriding in Java

Overriding provides subclasses with the ability to provide for its method implementation defined in its superclass.



```
@Override
protected void finalize() throws Throwable {
    try {
```

How To Override finalize() Method?

To override the **finalize()** method in a Java class, follow these steps:

- Declare a method named **finalize()** with the protected access modifier in your class. The protected modifier allows the method to be accessed by subclasses and classes within the same package.
- Add the @**Override** annotation above the method declaration to ensure you override the **finalize()** method from the superclass (which is **Object**).

- Specify that the method throws **Throwable**. This is required because the **finalize()** method throws **Throwable**, including **Exception** and **Error** subclasses.
- Implement the desired logic inside the **finalize()** method. This logic typically includes cleanup or finalization tasks that need to be performed before the object is garbage collected. Examples of such tasks include releasing resources, closing connections, or performing other cleanup operations.

Example:

```
public class MyClass {
    // Class members and methods go here
    @Override
    protected void finalize() throws Throwable {
        try {
            // Perform cleanup or finalization tasks here...
        } finally {
            super.finalize();
        }
    }
    public static void main(String[] args) {
        // Create an instance of MyClass
        MyClass myObject = new MyClass();
        // Perform some operations
     //Set the reference to null to make the object eligible for garbage
collection
        myObject = null;
        // Request garbage collection
        System.gc();
        // Perform some other operations
        // ...
    }
}
```

How Does the finalize() Method Works in Different Scenarios?

It's important to note that the **finalize ()** method is generally discouraged for critical resource cleanup or finalization tasks. It's recommended to use explicit resource management techniques, such as **try-with-resources**, to ensure proper resource release and cleanup.

Let us look at how **finalize ()** method works in two different scenarios:

- **Object without finalize() method overridden:**

If an object does not have the finalize () method overridden, the garbage collector will skip calling any specific finalization logic for that object. In this case, the object will still be eligible for garbage collection like any other object, but it won't have a chance to perform any custom cleanup tasks before being reclaimed.

- **Finalization and object resurrection:**

In some scenarios, an object can be "resurrected" during its finalization process. If an object's **finalize()** method resurrects the object by creating a new strong reference to it or adding it to some reachable data structure, the object will become reachable again and will not be garbage collected. This scenario is generally discouraged, as it can lead to unpredictable behavior and interfere with the normal garbage collection process.

Lifetime of the Finalized Object in Java

The lifetime of your object in Java employing **finalize()** method passes through several stages. These stages are as follows:

- Your object attains the "live" phase once you have created it.
- Once your object gets no references from the program, it is on the way to being reclaimed by the garbage collector.
- When the garbage collector identifies your object, it calls upon the **finalize()** method before reclaiming it.
- Upon completion of the **finalize()** method, the object is released from the memory and translocated into the "unreachable" phase.

Avoiding Finalizers

Disadvantage of Finalizers

The disadvantages of Finalizers are as follows:

- You cannot assign a particular time to call for this method. Hence it can get potentially risky to handle sophisticated operations.
- You can often face issues regarding performance
- It can be difficult at times for you to understand the lifetime of your object embedded in a program

Alternatives for finalize()

Some of the alternatives to **finalize()** method are as follows:

- You can use the **shut-down hook** to perform any cleanup processing.
- You can use a **reference queue** to release objects by performing cleanups when they become obsolete.
- You can use the **try-with-resources** statement, which came into existence in Java 7. This statement helps you to easily handle resources that need to be released when they become obsolete.
- You can use the **PhantomReference** class, which has a similar action to the previously mentioned alternatives.