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.

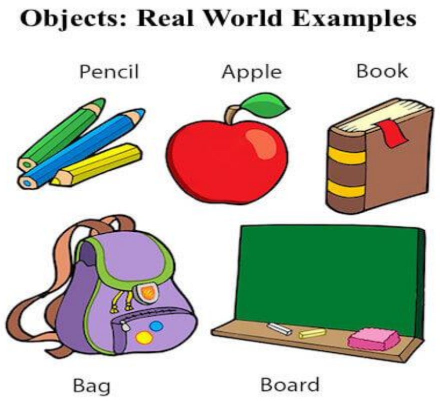
```
class Lamp {
    // stores the value for light
  // true if light is on
  // false if light is off
  boolean isOn;
  // method to turn on the light
  void turnOn() {
    isOn = true;
    System.out.println("Light on? " + isOn);
  }
  public static void main(String[] args) {
    // create an object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

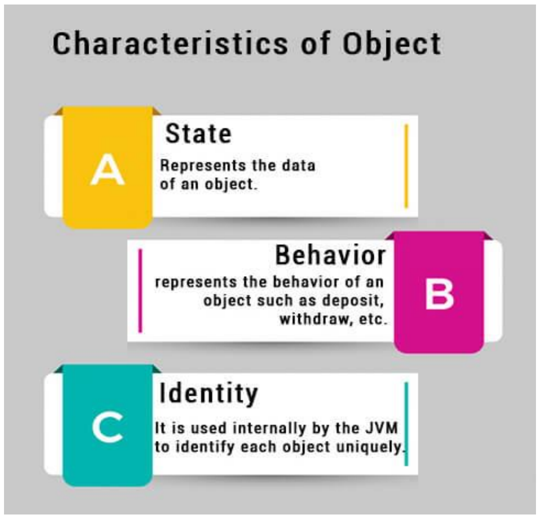
What is an object in Java



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.
B
Behavior
represents the behavior of an object such as deposit, withdraw, etc.
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

```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:

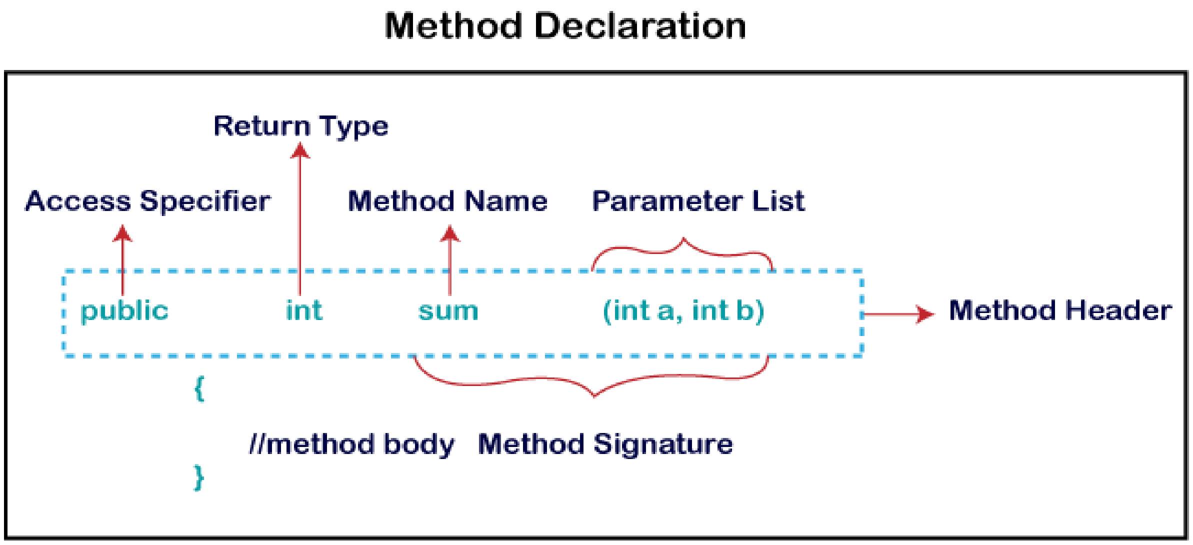
```
101 SE 45000
102 TE 25000
103 BE 55000
```

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 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
```

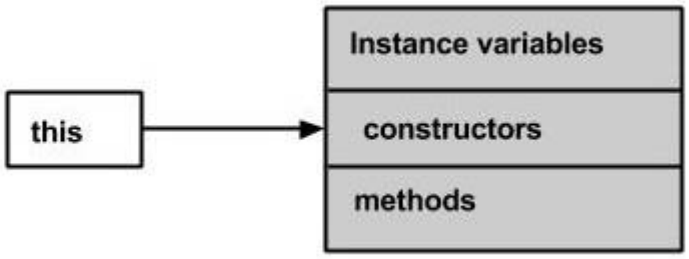
Output 2:

```
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?

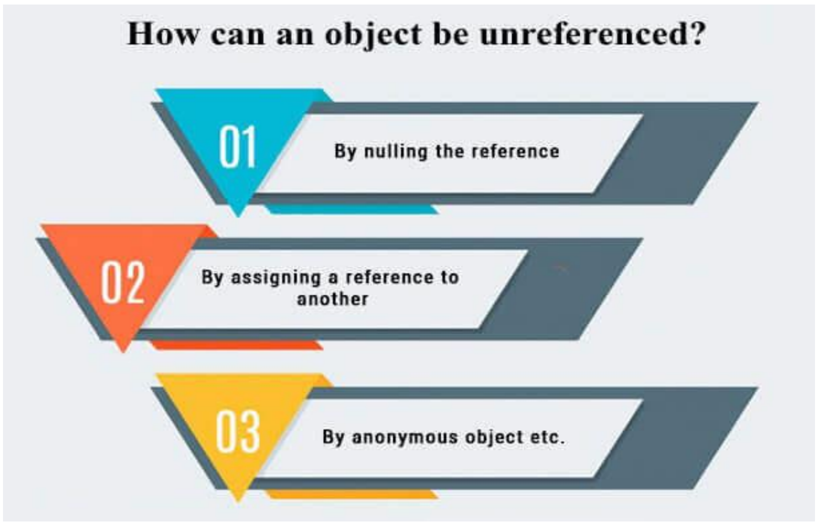
- Garbage collection in Java is the automated process of deleting code that's no longer needed or used.
- This automatically frees up memory space and ideally makes coding Java apps easier for developers.
- Java applications are compiled into bytecode that may be executed by a JVM.
- Objects are produced 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

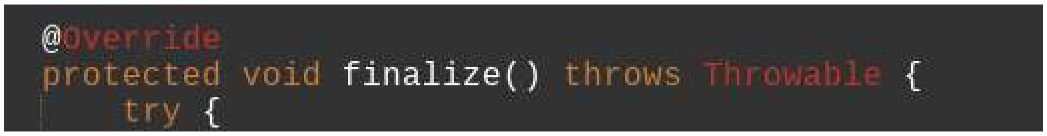
Why finalize() Method is used?

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 translocate 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.

Method Overloading in Java:

What is method overloading

- In Java, method overloading, as we discussed earlier, is a part of the polymorphism concept.
- This feature allows a class to have more than one method with the same name, as long as the parameters are different.
- The difference in parameters can be in terms of the number of parameters or the type of parameters.
- This provides the flexibility to call a similar method for different types of data.

Example of Method Overloading

- Let's have a simple example to illustrate method overloading. Say, we're about to program a calculator and the first step is to create a numbers addition function.
- Now, we could conjure a method named 'add' for two integers... or, just for a twist, we could whip up another 'add' for doubles, to gracefully handle those tricky non-integer numbers.
- Also let's create a method 'add' to add three integer numbers... just for our idea of overloading to demonstrate.
- Here's how it might look in our Java:

```java
public class OverloadingExample
{
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }
    // Overloaded method to add two double values
    public double add(double a, double b) {
        return a + b;
    }
    // Overloaded method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }
    public static void main(String[] args) {
        OverloadingExample example = new OverloadingExample();
        // Using the first add method
        System.out.println("Sum of two integers: " + example.add(10, 20));
```

```
            // Using the second add method
            System.out.println("Sum of two doubles: " + example.add(10.5, 20.5));
            // Using the third add method
            System.out.println("Sum of three integers: " + example.add(10, 20, 30));
        }
}
```

The output of this program is:

```
Sum of two integers: 30
Sum of two doubles: 31.0
Sum of three integers: 60
```

This is how method overloading is implemented in Java, allowing methods to be called with different types and numbers of arguments.

Method Overloading vs Method Overriding – What's the Difference?

- In Java, besides method overloading, there is also method overriding, which is very similar to it.
- The latter is also related to polymorphism, but they serve different purposes and have different rules. It is also connected with another important OOP concept - inheritance.
- So method overriding is a feature that allows a subclass to provide a specific implementation for a method that is already defined in its parent class.
- This is a fundamental part of the inheritance concept in object-oriented programming, allowing for runtime polymorphism.
- Here is a small code example. Let's create a class 'MusicalInstrument' and two subclasses: 'Piano' and 'Violin'.
- Both instruments are meant for playing, but the playing technique differs for each.
- We will override the parent method play () in both subclasses to demonstrate the concept.

```
class MusicalInstrument {
 void play() {
 System.out.println("Playing a musical instrument");
 }
}

class Violin extends MusicalInstrument {
 @Override
 void play() {
 System.out.println("Playing the violin");
 }
}

class Piano extends MusicalInstrument {
 @Override
 void play() {
 System.out.println("Playing the piano");
 }
}
public class Main {
 public static void main(String[] args) {
 MusicalInstrument myViolin = new Violin();
 MusicalInstrument myPiano = new Piano();

 myViolin.play(); // Outputs: Playing the violin
 myPiano.play(); // Outputs: Playing the piano
 }
}
```

- In both subclasses (Violin and Piano), the play method is overridden to provide specific behavior for each instrument.
- You can see the @Override annotation. It’s not a necessity, but it indicates that the method overrides a method of the superclass.
- At runtime, the JVM determines which specific play method to call based on the actual object type. It is called dynamic polymorphism.
- At the same time, method overloading represents Compile-time Polymorphism.
- Overloading is resolved during compile time, which is why it's also known as static polymorphism.

Let's sum it up, what's the difference between Method Overloading and Method Overriding:

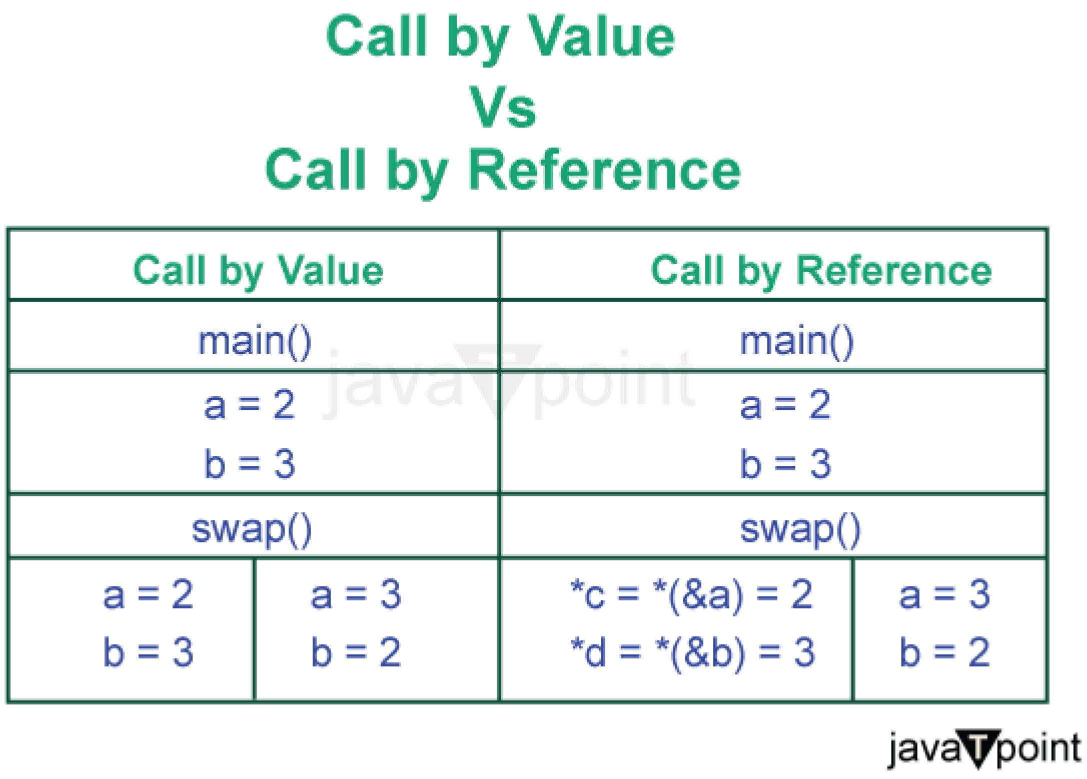
- Overloading is done within the same class; overriding involves a subclass and its superclass.
- In overloading, methods must have different parameters. In overriding, the method signature (name and parameters) must be the same.
- Overloaded methods are accessed based on the reference type, whereas overridden methods are accessed based on the object type.
- In method overriding, you can call the parent class method in the overriding method using the **super** keyword. This is not applicable in overloading.
- Compile-time Polymorphism is realized in overloading, while overriding is determined at runtime.

Parameter Passing Techniques in Java with Examples

Parameter passing in Java refers to the mechanism of transferring data between methods or functions. Java supports two types of parameters passing techniques

1. Call-by-value
2. Call-by-reference.

Understanding these techniques is essential for effectively utilizing method parameters in Java.



Call by Value Vs Call by Reference

Call by Value		Call by Reference	
main()		main()	

a = 2
b = 3
		a = 2
		b = 3	

swap()		swap()	

a = 2
b = 3
	a = 3
	b = 2
		*c = *(&a) = 2
		*d = *(&b) = 3
			a = 3
			b = 2

javaTpoint

Types of Parameters:

1. Formal Parameter:

- A variable and its corresponding data type are referred to as formal parameters when they exist in the definition or prototype of a function or method.
- As soon as the function or method is called and it serves as a placeholder for an argument that will be supplied.
- The function or method performs calculations or actions using the formal parameter.

Syntax:

```
returnType functionName(dataType parameterName)
{
   // Function body
   // Use the parameterName within the function
}
```

In the above syntax:

- returnType represents the return type of the function.
- functionName represents the name of the function.
- dataType represents the data type of the formal parameter.
- parameterName represents the name of the formal parameter.

2. Actual Parameter:

The value or expression that corresponds to a formal parameter and is supplied to a function or method during a function or method call is referred to as an actual parameter is also known as an argument. It offers the real information or value that the method or function will work with.

Syntax:

```
functionName(argument)
```

In the above syntax:

- functionName represents the name of the function or method.
- argument represents the actual value or expression being passed as an argument to the function or method.

1. Call-by-Value:

- In Call-by-value the copy of the value of the actual parameter is passed to the formal parameter of the method.
- Any of the modifications made to the formal parameter within the method do not affect the actual parameter.

ALGORITHM:

Step 1: Create a class named CallByValueExample.
Step 2: Inside the main method:
Step 2.1: Declare an integer variable num and assign it the value 10.
Step 2.2: Print the value of num before calling the method.
Step 2.3: Call the modifyValue method, passing num as the actual parameter.
Step 2.4: Print the value of num after calling the method.
Step 3: Define the modifyValue method that takes an integer parameter value:
Step 3.1: Modify the formal parameter value by assigning it the value 20.
Step 3.2: Print the value of value inside the method.

Implementation:

The implementation of the above steps given below:

FileName: CallByValueExample.java

```java
import java.util.*;
public class CallByValueExample
{
   public static void main(String[] args)
   {
      int num = 10;
      System.out.println("Before calling method:"+num);
      modifyValue(num); // Calling the method and passing the value of 'num'
      System.out.println("After calling method:"+num);
   }
   public static void modifyValue(int value) {

      // Modifying the formal parameter
      value=20; // Assigning a new value to the formal parameter as value
      System.out.println("Inside method:"+value);
   }
}
```

Output:

```
Before calling method: 10
Inside method: 20
After calling method: 10
```

Call-by-Reference:

- “call by reference" is a method of passing arguments to functions or methods where the memory address (or reference) of the variable is passed rather than the value itself.
- This means that changes made to the formal parameter within the function affect the actual parameter in the calling environment.
- In "call by reference," when a reference to a variable is passed, any modifications made to the parameter inside the function are transmitted back to the caller.
- This is because the formal parameter receives a reference (or pointer) to the actual data.

ALGORITHM:

- **Step 1:** Start
- **Step 2:** Define the class "CallByReference"
- **Step 2.1:** Declare instance variables: a (int) and b (int)
- **Step 2.1:** Define a constructor to assign values to a and b
- **Step 3:** Define the method "changeValue" inside the "CallByReference" class:
- **Step 3.1:** Accept a parameter of type "CallByReference" called "obj"
- **Step 3.2:** Add 10 to the value of "obj.a"
- **Step 3.3:** Add 20 to the value of "obj.b"
- **Step 4:** Define the class "Main"
- **Step 4.1:** Define the main method
- **Step 4.2:** Create an instance of "CallByReference" called "object" with values 10 and 20
- **Step 4.3:** Print the values of "object.a" and "object.b"
- **Step 4.4:** Call the "changeValue" method on "object" and pass "object" as an argument
- **Step 4.5:** Print the updated values of "object.a" and "object.b"
- **Step 5:** End

Implementation:

The implementation of the above steps given below

FileName: CallByReferenceExample.java

```
import java.util.*;
class CallByReference   // Callee
{
    int a,b;
    // Constructor to assign values to the class variables
    CallByReference(int x,int y)
    {
        a=x;
        b=y;
    }
   // Method to change the values of class variables
    void changeValue(CallByReference obj)
    {
        obj.a+=10;
        obj.b+=20;
    }
}
public class CallByReferenceExample   // Caller
{
    public static void main(String[] args)
    {
        // Create an instance of CallByReference and assign values using the constructor
        CallByReference object=new CallByReference(10, 20);
        System.out.println("Value of a: "+object.a +" & b: " +object.b);
        // Call the changeValue method and pass the object as an argument
        object.changeValue(object);

        // Display the values after calling the method
        System.out.println("Value of a:"+object.a+ " & b: "+object.b);
    }
}
```

Output:

```
Value of a: 10 & b: 20
Value of a: 20 & b: 40
```

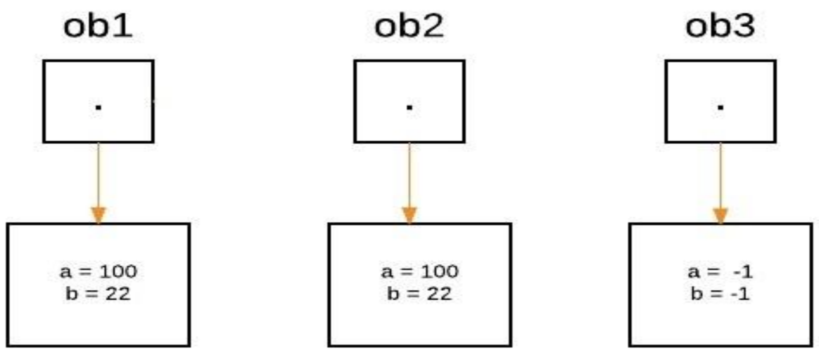
Passing and Returning Objects in Java

- Although Java is strictly passed 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.
- Basically, a parameter cannot be changed by the function, but the function can ask the parameter to change itself via calling some method within it.

  - ➢ 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 the object used as an argument.

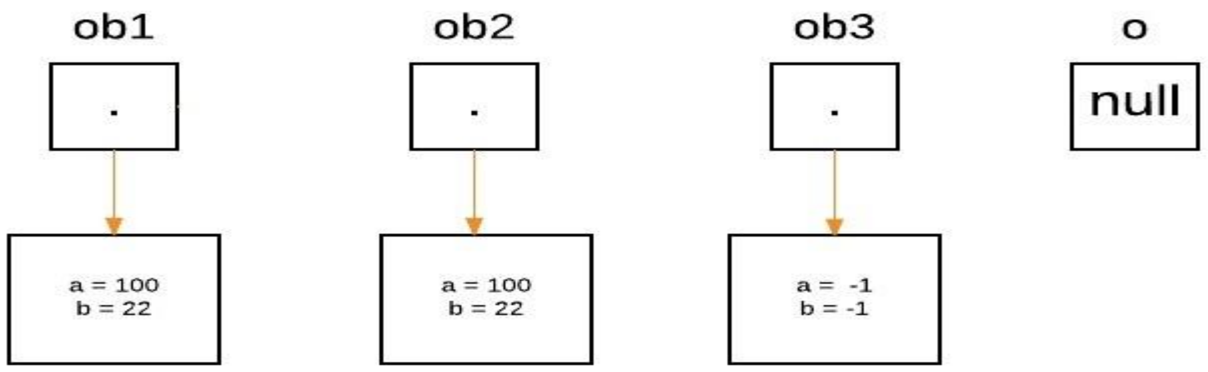
Illustration: Let us suppose three objects 'ob1' , 'ob2' and 'ob3' are created:

```
ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);

ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);

ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);
```



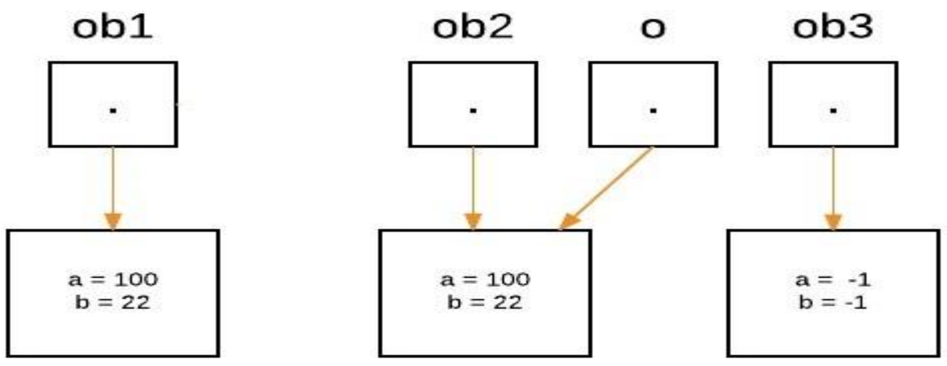
From the method side, a reference of type Foo with a name a is declared and it's initially assigned to null.

boolean equalTo(ObjectPassDemo o);



As we call the method equalTo, the reference 'o' will be assigned to the object which is passed as an argument, i.e. 'o' will refer to 'ob2' as the following statement execute.

System.out.println ("ob1 == ob2: " + ob1.equalTo (ob2));

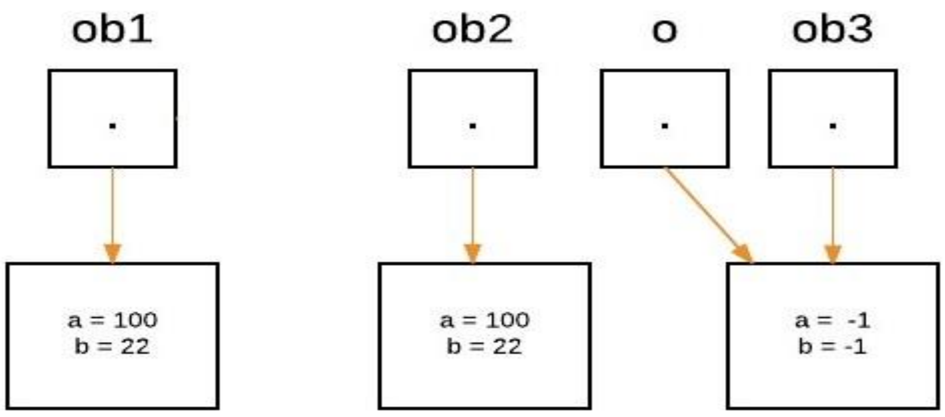


Now as we can see, equalTo method is called on 'ob1', and 'o' is referring to 'ob2'. Since values of 'a' and 'b' are same for both the references, so if (condition) is true, so boolean true will be return.

if (o.a == a && o.b == b)

Again 'o' will reassign to 'ob3' as the following statement execute.

System.out.println ("ob1 == ob3: " + ob1.equalTo (ob3));



- Now as we can see, the equalTo method is called on 'ob1' , and 'o' is referring to 'ob3'. Since values of 'a' and 'b' are not the same for both the references, so if(condition) is false, so else block will execute, and false will be returned.
- In Java we can pass objects to methods as one can perceive from the below program as follows:

Example:

```
// Java Program to Demonstrate Objects Passing to Methods.
// Class     // Helper class
class ObjectPassDemo {
        int a, b;

        // Constructor
        ObjectPassDemo(int i, int j)
        {
                a = i;
                b = j;
        }
        // Method
        boolean equalTo(ObjectPassDemo o)
        {
                // Returns true if o is equal to the invoking
                // object notice an object is passed as an
                // argument to method
                return (o.a == a && o.b == b);
        }
}
```

```
// Main class
public class GFG {
        // Main driver method
        public static void main(String args[])
        {
                // Creating object of above class inside main()
                ObjectPassDemo ob1 = new ObjectPassDemo(100, 22);
                ObjectPassDemo ob2 = new ObjectPassDemo(100, 22);
                ObjectPassDemo ob3 = new ObjectPassDemo(-1, -1);

// Checking whether object are equal as custom values above passed and printing
// corresponding boolean value
                System.out.println("ob1 == ob2: "
                                                + ob1.equalTo(ob2));
                System.out.println("ob1 == ob3: "
                                                + ob1.equalTo(ob3));
        }
}
```

Output

```
ob1 == ob2: true

ob1 == ob3: false
```

Defining a constructor that takes an object of its class as a parameter

- One of the most common uses of object parameters involves constructors.
- Frequently, in practice, there is a need to construct a new object so that it is initially the same as some existing object.
- To do this, either we can use Object.clone() method or define a constructor that takes an object of its class as a parameter.

Example:

```
// Java program to Demonstrate One Object to Initialize Another
// Class 1
class Box {
        double width, height, depth;
// Notice this constructor. It takes an object of type Box.
//This constructor use one object to initialize another
Box(Box ob)
        {
                width = ob.width;
                height = ob.height;
                depth = ob.depth;
        }
// constructor used when all dimensions specified
        Box(double w, double h, double d)
        {
                width = w;
                height = h;
                depth = d;
        }
// compute and return volume
        double volume() { return width * height * depth; }
}

// Main class
public class GFG {
        // Main driver method
        public static void main(String args[])
        {
                // Creating a box with all dimensions specified
                Box mybox = new Box(10, 20, 15);
                // Creating a copy of mybox
                Box myclone = new Box(mybox);
                double vol;
                // Get volume of mybox
                vol = mybox.volume();
                System.out.println("Volume of mybox is " + vol);
                // Get volume of myclone
                vol = myclone.volume();
                System.out.println("Volume of myclone is " + vol);
        }
}
```

Output

```
Volume of mybox is 3000.0

Volume of myclone is 3000.0
```

Returning Objects

- In java, a method can return any type of data, including objects.
- For example, in the following program, the incrByTen( ) method returns an object in which the value of an (an integer variable) is ten greater than it is in the invoking object.

Example:

```
// Java Program to Demonstrate Returning of Objects
// Class 1
class ObjectReturnDemo {
        int a;
        // Constructor
        ObjectReturnDemo(int i) { a = i; }
        // Method returns an object
        ObjectReturnDemo incrByTen()
        {
                ObjectReturnDemo temp
                        = new ObjectReturnDemo(a + 10);
                return temp;
        }
}
// Class 2   // Main class
public class GFG {
        // Main driver method
        public static void main(String args[])
        {
                // Creating object of class1 inside main() method
                ObjectReturnDemo ob1 = new ObjectReturnDemo(2);
                ObjectReturnDemo ob2;
                ob2 = ob1.incrByTen();
                System.out.println("ob1.a: " + ob1.a);
                System.out.println("ob2.a: " + ob2.a);
        }
}
```

Output

```
ob1.a: 2

ob2.a: 12
```

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.

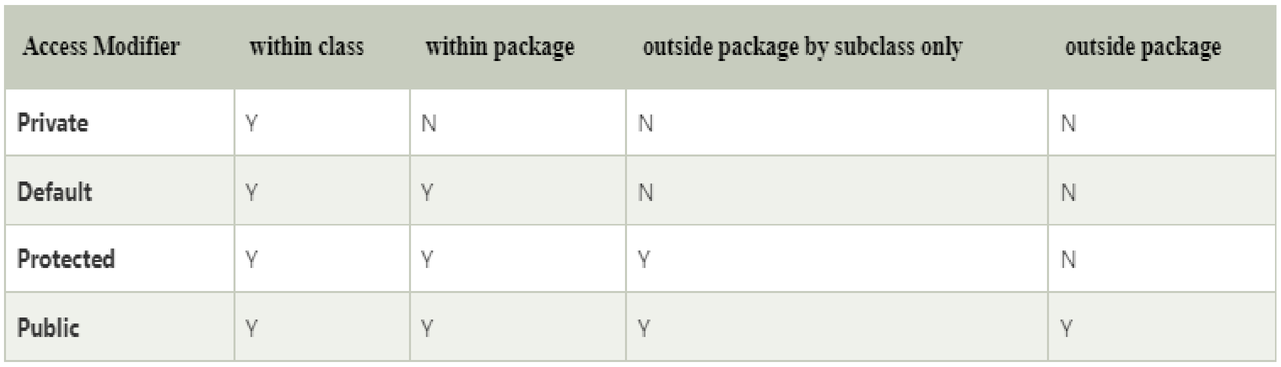
There are four types of Java access modifiers:

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.

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

1) 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
   }
}
```

Role of Private Constructor

If you make any class constructor private, you cannot create the instance of that class from outside the class. For example:

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

Note: A class cannot be private or protected except nested class.

2) 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.

```
//A.java
package pack;
class A{
  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();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.

3) 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 modifier.

Example of protected access modifier

In this example, we have created the two packages pack and mypack. The A class of pack package is public, so can be accessed from outside the package. But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

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

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

Output:Hello

4) Public

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

Example of public access modifier

```
//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();
  }
}
```

Output:Hello

Java Access Modifiers with Method Overriding

If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.

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

public class Simple extends A{
void msg(){System.out.println("Hello java");}//C.T.Error
 public static void main(String args[]){
   Simple obj=new Simple();
   obj.msg();
   }
}
```

The default modifier is more restrictive than protected. That is why, there is a compile-time error.

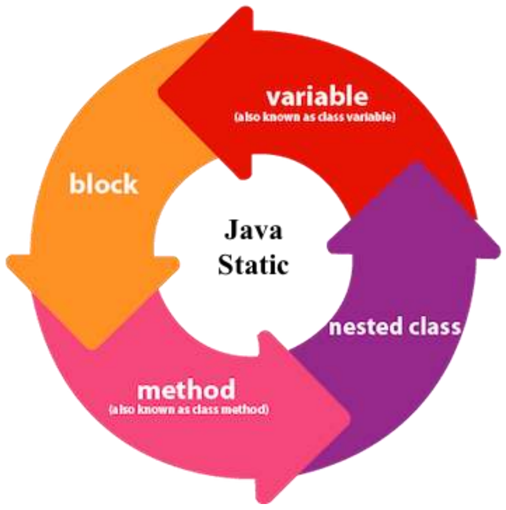
Java static keyword

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:

1. Variable (also known as a class variable)
2. Method (also known as a class method)
3. Block
4. Nested class

1) 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;
     String college="ABC";
}
```

Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

Java static property is shared to all objects.

Example of static variable

```
//Java Program to demonstrate the use of static variable
class Student{
   int rollno;//instance variable
   String name;
   static String college ="MESWCOE";//static variable
   //constructor
   Student(int r, String n){
   rollno = r;
   name = n;
   }
   //method to display the values
   void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"ABC");
 Student s2 = new Student(222,"XYZ");
 //we can change the college of all objects by the single line of code
 //Student.college="Wadia";
 s1.display();
 s2.display();
 }
}
```

Output:

```
111 ABC MESWCOE
222 XYZ MESWCOE
```

Program of the counter without static variable

In this example, we have created an instance variable named count which is incremented in the constructor. Since instance variable gets the memory at the time of object creation, each object will have the copy of the instance variable. If it is incremented, it won't reflect other objects. So each object will have the value 1 in the count variable.

```
//Java Program to demonstrate the use of an instance variable
//which get memory each time when we create an object of the class.
class Counter{
int count=0;//will get memory each time when the instance is created

Counter(){
count++;//incrementing value
System.out.println(count);
}
public static void main(String args[]){
//Creating objects
Counter c1=new Counter();
Counter c2=new Counter();
Counter c3=new Counter();
}
}
```

Output:

```
1
1
1
```

```
//Java Program to illustrate the use of static variable which is shared with all objects.
class Counter2{
static int count=0; //will get memory only once and retain its value

Counter2(){
count++;//incrementing the value of static variable
System.out.println(count);
}

public static void main(String args[]){
//creating objects
Counter2 c1=new Counter2();
Counter2 c2=new Counter2();
Counter2 c3=new Counter2();
}
}
```

Output:

```
1
2
3
```

2) 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.

Example of static method

```
//Java Program to demonstrate the use of a static method.
class Student{
    int rollno;
    String name;
    static String college = "MESWCOE";
    //static method to change the value of static variable
    static void change(){
    college = "Wadia";
    }
    //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,"ABC");
   Student s2 = new Student(222,"XYZ");
   Student s3 = new Student(333,"PQR");
   //calling display method
   s1.display();
   s2.display();
   s3.display();
   }
}
```

```
Output:
111 ABC Wadia
222 XYZ Wadia
333 PQR Wadia
```

Another example of a static method that performs a normal calculation

```
//Java Program to get the cube of a given number using the static method
class Calculate{
 static int cube(int x){
 return x*x*x;
 }
 public static void main(String args[]){
 int result=Calculate.cube(5);
 System.out.println (result);
 }
}
```

```
Output:
125
```

Restrictions for the static method

There are two main restrictions for the static method. They are:

1. The static method cannot use non static data member or call non-static method directly.
2. this and super cannot be used in static context.

```
class A{
 int a=40;//non static

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

```
Output:
Compile Time Error
```

3) 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");
 }
}
```

```
Output:
Static block is invoked
Hello main
```

Nested Class in Java

- In Java, just like nested Loops, we have **Nested Classes**, i.e., we can define a class inside another class.
- We can group classes with similar properties under an outer class, so they are together.
- It **increases the encapsulation and readability** of the code. The nested class comes under the principles of Object-Oriented Programming (OOP).
- We cannot define normal classes as private. But Java treats the nested class as members of the outer class, and we can define them as private.

Properties of Nested classes in Java

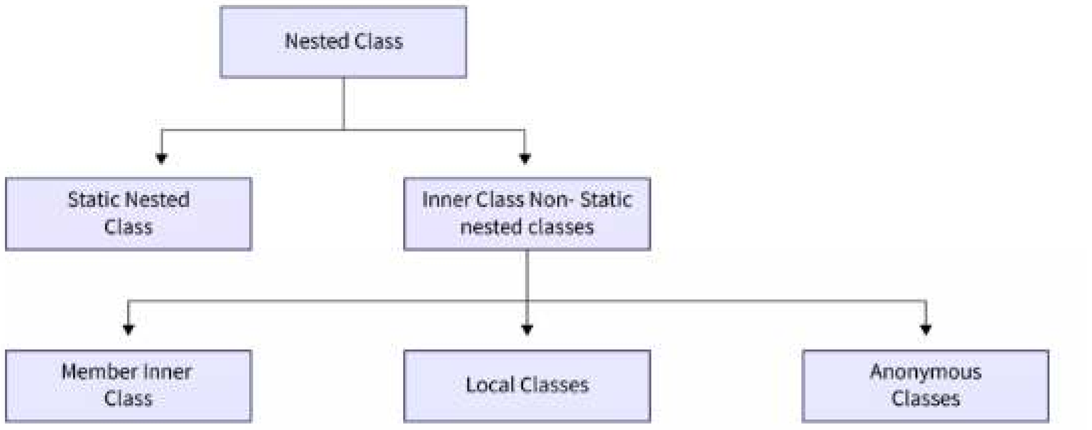
- In Java, a nested class is a class that is defined inside some other class.
- Nested classes are used to group certain classes to improve the readability of the code.
- The scope of a nested class is the same as its outer class.
- Nested classes can access any member of the outer class, even if the outer class is private.

Note: Outer class does not have access to the Inner class if the inner class is private. We use nested classes to group into different classes and to increase security by limiting the scope of classes.

Types of Nested classes in Java

A nested class can either be defined as a static type or a non-static type. Types in detail:-

- Static Nested Classes
- Non-static Nested Classes (or Inner Classes) We can further divide the Non-static Classes into -
  - Member Inner Classes
  - Local Inner Classes
  - Anonymous Inner Classes



Static Nested Classes

- A static class is a class that is created inside a class. A static variable is common to all the instances of that particular class.
- Similarly, a static class can access all the instance functions of the outer class.
- To do this, we will have to make objects of child classes and refer to them. We can access all the static functions of the outer class without object creation.

Syntax:

```
class parent_class
{
   static class static_child_class
   {
      // code
   }
}
```

- We can create an object of this class like this:

```
static_child_class obj = new static_child_class();
```

Let us see an example of using the static nested class and accessing its method from other functions.

```
// parent class
class parent_class {
   static String s = "Wadia";
   // child class
   static class static_child_class {
      // child class method
      void print(String x) {
         System.out.println(s + " " + x);
      }
   }
   public static void main(String args[]) {
      // child class object
      static_child_class obj = new static_child_class();
      String y = " College ";
      obj.print(y);
   }
}
```

Output:

```
Wadia College
```

Explanation: Here, we have created a nested class. The parent_class is the outer class, and the static_child_class is the inner class. The main() function of the parent_class access the inner class methods with the object.

Non-static Nested Classes (Inner Classes)

- A non-static nested class or inner class is a class within a class. We do not define it as static so that it can directly use all the functions and variables of the outer class.
- From the inner class, if we want to access any static method of the outer class, we do not need any object; we can call it directly.

1. Member Inner Classes

Syntax:

```
class parent_class {
   class child_class {
      // code
   }
}
```

- To access the non-static inner class from the static main method or any other static method, we have to create objects of the parent_class as well as the child_class.
- The syntax for these objects:

```
parent_class parentObj = new parent_class();
child_class childObj = parentObj.new child_class();
```

Let us see an example of using the non-Static nested class and accessing its method from other functions:-

```
// parent class
class parent_class {
   String s = "Wadia";
   // child class
   class child_class {
      void print(String x) {
         System.out.println(s + " " + x);
      }
   }
   public static void main(String args[]) {
      // parent class object
      parent_class parentObj = new parent_class();
      // child class object using parent class object
```

```
        child_class childObj = parentObj.new child_class();
        String y = "Topics";
        // calling methods of child class
        childObj.print(y);
    }
}
```

Output:

```
Wadia College
```

Explanation: Here, we are creating an object of the parent class and, with the help of it, creating the object of the child class. We are accessing the methods of the child_class using the childObj. We had to do this because the child_class is non-static, while the main method is static.

2. Local Inner Classes

A Local Inner class is a class that is defined inside any **block**, i.e., for block, if block, methods, etc. Similar to local variables, **the scope of the Local Inner Class is restricted to the block where it is defined**.

Syntax:

```
class class_name {
    void method_name() {
        // code
        if(conditions) {
        // or any other block like while, for, etc.
            class localInnerClass {
                void localInnerMethod() {
                    // code
                }
            }
        }
        // code
    }
}
```

Let us see an example of using the Local inner class and accessing its method:-

```
// parent class
class parent_class {
    public static void main(String args[]) {
        String s = "Wadia";
        if (s.charAt(0) == 'W') {
            // child class
            class child_class {
                void print(String x) {
                    System.out.println(s + " " + x);
                }
            }
            // child class object
            child_class childObj = new child_class();
            String y = " College ";
            // calling child class method
            childObj.print(y);
            // child_class is accessible till here only
        }
        // child_class is not accessible here
    }
}
```

Output:

```
Wadia College
```

Explanation: Here, we are defining a class inside an "if block". And we are creating an object of child_class and calling it from the same block. The child_class is only accessible from the if block and cannot be called from outside.

3. Anonymous Inner Classes

- Anonymous Inner class is an **inner class but without a name**. It has only a single object. It is used to override a method.
- It is only accessible in the block where it is defined.
- We can use an abstract class to define the anonymous inner class. It has access to all the members of the parent class. It is useful to shorten over code.
- Basically, **it merges the step of creating an object and defining the class.**

Syntax:

```
abstractClass obj = new abstractClass() {
    void methods() {
        // code
    }
};
```

Let us see an example of Anonymous inner class and accessing its methods:-

```
// abstract Class
abstract class Printer  {
   abstract void print(String x);
}
// Parent Class
class parent_class {
   public static void main(String args[]) {
      // Anonymous Inner Class
      Printer obj = new Printer() {
         void print(String x) {
            System.out.println("Wadia " + x);
         }
      };
      String y = "Topics";
      obj.print(y);
   }
}
```

Output:

```
Wadia College
```

Explanation: The Printer object is used to define the anonymous inner class. Then we are calling the print() method of the anonymous inner class using the object we just defined.

Conclusion

- Nested classes in Java are classes inside another class.
- We use Nested Classes to group into different classes and to **increase the security** by limiting the scope of classes.
- While compiling a Nested class, multiple **.class** files are generated, 1 for each class.
- In Nested classes, the outer class does not have access to the data and methods of the inner class if the inner class is private.
- We have two types of Nested Classes
    - Static Nested Class
    - Non-Static Nested Class

What are the command line arguments in Java?

The command line arguments in java are the arguments passed to the program from the console.

Overview

- **Command line Argument** is a way to provide input to Java applications through the command line when compiling and running the program. These are the inputs provided after the execution command when running a Java program. They are passed to the main function as strings and stored in the args parameter.
- **Command line arguments** are processed by the **Java Virtual Machine (JVM)** before the main function is invoked. They are bundled together and supplied to the main function as an array of strings. Command line arguments are read as strings, and certain characters like #, %, and & may cause errors when entered as command line arguments.

Working of Command-line Arguments

There are various ways to provide input to our program. One of these ways is to provide input through the command line, which we also use to compile and run our program. These inputs are passed to the main function at runtime even before our main () begins to execute. The image below represents the two ways by which we can write our main function in Java. The argument String [] args or String… args can be broken down as follows.
In the context of command line arguments in Java:

- **String**: It represents the data type of the array elements passed to the main function. In Java, command line arguments are received as an array of strings. The String data type is used because Java commonly uses strings for input and output operations. Each command line argument is treated as a string, regardless of its original data type.
- **args**: The local name given to the array variable holds the command line arguments in the main function. It serves as a reference to the array, allowing access to and modifying the values stored in it. Using the args variable, the program can process and manipulate the command line arguments as needed during runtime.

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

The command line arguments that are given to the program are stored in these main() function parameters.

Java programs are compiled and run by:

Compile > javac filename.java

Run > java filename

- To execute our program, we need to compile the .java file using the above-stated command. This converts our .java file to a .class file in a machine-readable language. Later, we execute this file using the run command.
- If we provide inputs following this command, the JVM (Java Virtual Machine) wraps all these inputs together and supplies them to the main function parameter String[] args or String... args.
- This process occurs even before our main() is invoked.
- These inputs are referred to as command-line arguments. Since these inputs are passed to our application before its execution, they provide us with a way to configure our application with different parameters.
- This capability is particularly useful for efficient testing of our application.

To pass command line arguments in Java, follow these steps:

1. Write your Java program and save it with the .java extension in your local system.
2. Open the command prompt (cmd) and navigate to the directory where you saved your program.
3. Compile your program using the command javac filename.java. This will convert the .java file to a .class file.
4. Write the run command java filename and continue the command with all the inputs you want to give to your code.
5. After writing all your inputs, run the command to execute your program.

- The command line arguments given to the program are bundled and supplied to the main function parameter even before the invoking of the main function.

- These inputs are stored in a string array since Java deals with strings when performing I/O operations.
- The inputs are distinguished by the blank space ' ' between them.
- The command line reads one character at a time and stores it.
- When it encounters a space, it saves the already collected text as an input value and repeats the process until all characters are read.

Simple Example of Command-line Argument in Java:

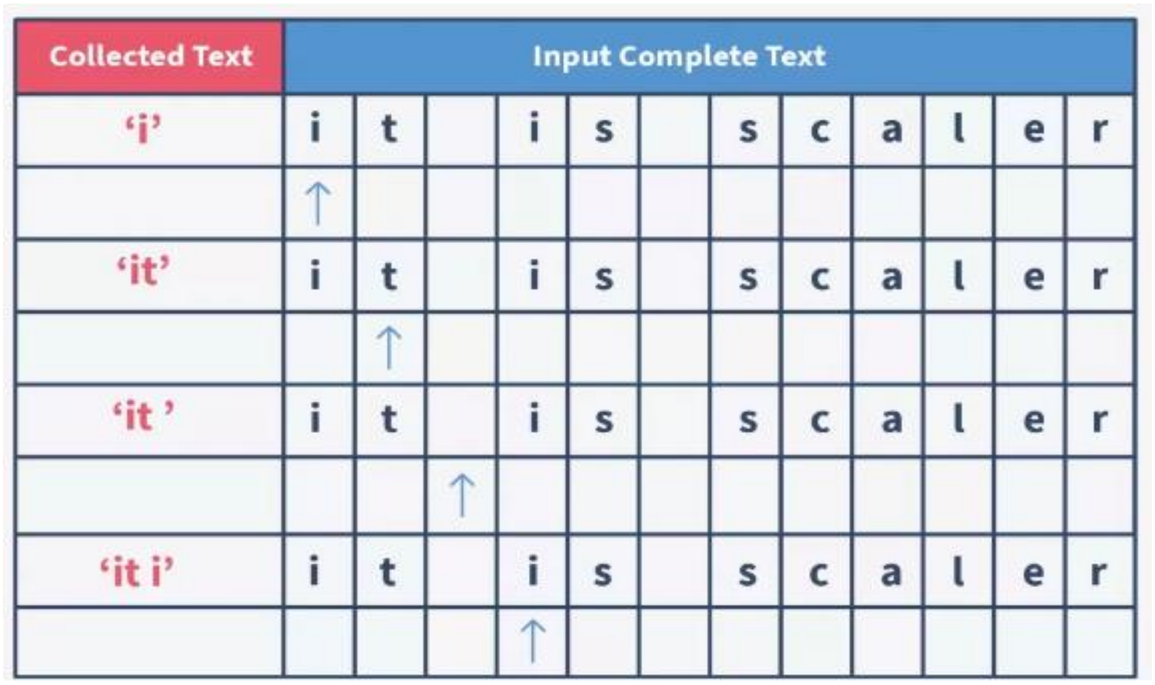
```
public class cmdArgs {
  public static void main(String[] args) {
    System.out.println(
      "The number of command line inputs entered are " + args.length
    );
    for (int i = 0; i < args.length; i++) System.out.println(
      "input " + i + " is " + args[i]
    );
  }
}
```

Output:

```
$ javac cmdArgs.java
$ java cmdArgs apple banana mango graes orange
The number of command line inputs entered is 5
input 1 is apple
input 2 is banana
input 3 is mango
input 4 is grapes
input 5 is orange
$
```

- The figure below gives a step-by-step walkthrough of how the command line inputs are read.
- As soon as the run command gets executed, the inputs following it are read as strings into the main function parameter.

- This is done in such a way that the JVM reads every string individually when it encounters a character.
- As it reads a space character, it saves the string and starts with another string.
- This restricts the user from entering sentences as a command line input.



Collected Text	Input Complete Text											
‘i’	i	t		i	s		s	c	a	l	e	r
	↑											
‘it’	i	t		i	s		s	c	a	l	e	r
		↑										
‘it ’	i	t		i	s		s	c	a	l	e	r
			↑									
‘it i’	i	t		i	s		s	c	a	l	e	r
				↑								

Variable Argument (Varargs):

- The varrags allows the method to accept zero or muliple arguments.
- Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem.
- If we don't know how many argument we will have to pass in the method, varargs is the better approach.

Advantage of Varargs:

We don't have to provide overloaded methods so less code.

Syntax of varargs:

The varargs uses ellipsis i.e. three dots after the data type. Syntax is as follows:

```
return_type method_name(data_type... variableName){}
```

Simple Example of Varargs in java:

```
class VarargsExample1{

 static void display(String... values){
  System.out.println("display method invoked ");
 }

 public static void main(String args[]){

 display();//zero argument
 display("my","name","is","varargs");//four arguments
 }
}
```

Output:

```
display method invoked

display method invoked
```

Another Program of Varargs in java:

```
class VarargsExample2{
       static void display(String... values){
       System.out.println("display method invoked ");
       for(String s:values){
        System.out.println(s);
       }
      }

      public static void main(String args[]){

      display();//zero argument
      display("hello");//one argument
      display("my","name","is","varargs");//four arguments
      }
     }
```

Output:

```
display method invoked
display method invoked
hello
display method invoked
my
name
is
varargs
```

Rules for varargs:

While using the varargs, you must follow some rules otherwise program code won't compile. The rules are as follows:

- There can be only one variable argument in the method.
- Variable argument (varargs) must be the last argument.

Examples of varargs that fails to compile:

```
void method(String... a, int... b){}//Compile time error

void method(int... a, String b){}//Compile time error
```

Example of Varargs that is the last argument in the method:

```
class VarargsExample3{

 static void display(int num, String... values){
  System.out.println("number is "+num);
  for(String s:values){
   System.out.println(s);
  }
 }

 public static void main(String args[]){

 display(500,"hello");//one argument
 display(1000,"my","name","is","varargs");//four arguments
 }
}
```

Output:

```
number is 500
hello
number is 1000
my
name
is
varargs
```