JAVA ARRAYS

JAVA ARRAYS [SINGLE DIMENTIONAL ARRAYS]

Let us see how to work with arrays in Java to declare, initialize, and access array elements with examples. **An array** is a collection of similar types of data. E.g., to store names of **1000 people,** we can create an array of a string type that can store **100 names**.

String[ ] array = new String[100];

Here, the above array cannot store more than 100 names. The number of values in a Java array is always fixed.

How to declare an array in Java?

In Java, you can declare an array as seen below.

dataType[ ] arrayName;

- **dataType** - it can be primitive data types like int, char, double, byte, etc. or Java objects
- **arrayName** - it is an identifier

For example,

double[ ] data;

Here, data is an array that can hold values of the double datatype. But, how many elements can this array hold? **Good question!** To define the number of elements that an array can hold, we have to **allocate memory** for that array in Java. For example,

```
// declare an array
double[ ] data;

// allocate memory
data = new double[10];
```

Here, the above array can store **10 elements**. We can also say that the **size** or **length** of the array is 10. In Java, we can ably **declare** and **allocate** the memory of **an array** in **one single statement**. For example,

double[ ] data = new double[10];

How to Initialize Arrays in Java?

In Java, we can initialize arrays during declaration. For example,

```
//declare and initialize and array
int[ ] age = {12, 4, 5, 2, 5};
```

Here, we have created an array named **age** and **initialized it** with the **values** inside the curly brackets. Note that we have not provided the size of the array. In this case, the Java compiler automatically specifies the size by counting the number of elements in the array (i.e. 5). In the

Java array, **each memory location** is associated with a number. The number is known as an **array index**. We can also initialize arrays in Java, using the **index number**. For example,

```
// declare an array
int[ ] age = new int[5];

// initialize array
age[0] = 12;
age[1] = 4;
age[2] = 5;
..
```

Elements are stored in the array
Java Arrays initialization
Note:

- **Array indices** always start from **0**. That is, the **first element** of an array is at **index 0**.
- If the size of an array is **n**, then the last element of the array will be at **index n-1**.

HOW TO ACCESS ELEMENTS OF AN ARRAY IN JAVA?

You can access **an array element** using its **index number**. Here is the syntax for accessing elements of an array,

```
// access array elements
array[index]
```

Let's see an example of accessing array elements using index numbers.

Example: Access Array Elements

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

  // create an array
  int[ ] age = {12, 4, 5, 2, 5};

  // access each array elements
  System.out.println("Accessing Elements of Array:");
  System.out.println("First Element: " + age[0]);
  System.out.println("Second Element: " + age[1]);
  System.out.println("Third Element: " + age[2]);
  System.out.println("Fourth Element: " + age[3]);
  System.out.println("Fifth Element: " + age[4]);
 } }
```

In the above example, notice that we are using the index number to access each element of the array. We can use **loops** to access all the elements of the array at once.

Looping Through Array Elements

In Java, we can also loop through each element of the array. For example,

Example: Using For Loop

```
class Main {
 public static void main(String[] args) {
  // create an array
  int[ ] age = {12, 4, 5};
  // loop through the array
  // using for loop
  System.out.println("Using for Loop:");
  for(int i = 0; i < age.length; i++) {
    System.out.println(age[i]);
  } } }
```

In the above example, we are using the **Java for Loop** to iterate through each element of the array. Notice the expression inside the loop, **age.length.** Here, we are using the length property of the array to get the size of the array.

We can also use the for-each loop to iterate through the elements of an array. For example,

Example: Using the for-each Loop

```
class Main {
 public static void main(String[] args) {
  // create an array
  int[ ] age = {12, 4, 5};
  // loop through the array
  // using for loop
  System.out.println("Using for-each Loop:");
  for(int a : age) {
    System.out.println(a);
  } } }
```

Example: Compute Sum and Average of Array Elements

```
class Main {
 public static void main(String[ ] args) {
  int[ ] numbers = {2, -9, 0, 5, 12, -25, 22, 9, 8, 12};
  int sum = 0;
  Double average;
  // access all elements using for each loop
  // add each element in sum
```

```
for (int number: numbers) {
  sum += number;
}
// get the total number of elements
int arrayLength = numbers.length;
// calculate the average
// convert the average from int to double
average =  ((double)sum / (double)arrayLength);
System.out.println("Sum = " + sum);
System.out.println("Average = " + average);
}}
```

In the above example, we have created **an array** of named numbers and used the **for...each loop** to access each element of the array. Inside the loop, we are calculating the sum of each element.

Multidimensional Arrays

We can also declare multidimensional arrays in Java. **A multidimensional array** is an array of arrays. That is, every element of a multidimensional array is an array itself. For example,

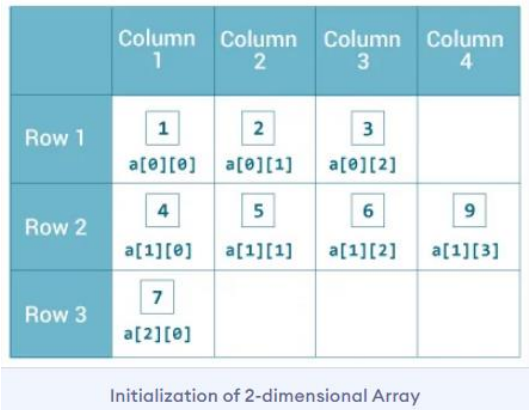
```
double[ ][ ] matrix = {{1.2, 4.3, 4.0},
    {4.1, -1.1}
};
```

Here, we have created a multidimensional array named matrix which is a 2-dimensional array. **Multidimensional arrays** include the **2-dimensional arrays** and the **3-dimensional arrays**. Before you learn about the **multidimensional array**, make sure you know about Java array. A **multidimensional array** is an array of arrays. Every multidimensional array element is an array

```
int[ ][ ] a = new int[3][4];
```

Here, we have created a **multidimensional array** named **a**. It is a 2-dimensional array, that can hold a maximum of 12 elements,

2-dimensional array in Java



Initialization of 2-dimensional Array

Remember, Java uses zero-based indexing, that is, indexing of arrays in Java starts with 0 and not 1. Let's take another example of the multidimensional array. This time we will be creating a **3-dimensional array**. For example,

```
String[ ][ ][ ] data = new String[3][4][2];
```

Here, data is a 3d array that can hold a maximum of 24 (3*4*2) elements of type String.

Here is how we can initialize a 2-dimensional array in Java.

```
int[ ][ ] a = {
    {1, 2, 3},
    {4, 5, 6, 9},
    {7},
};
```

As we can see, **each element** of the multidimensional array is an array itself. And also, unlike C/C++, each row of the **multidimensional array** in Java can be of different lengths.

2d array example in Java with variable length
Initialization of 2-dimensional Array
Example: 2-dimensional Array

```
class MultidimensionalArray {
   public static void main(String[ ] args) {
      // create a 2d array
      int[ ][ ] a = {
         {1, 2, 3},
         {4, 5, 6, 9},
         {7},
      };
      // calculate the length of each row
      System.out.println("Length of row 1: " + a[0].length);
      System.out.println("Length of row 2: " + a[1].length);
      System.out.println("Length of row 3: " + a[2].length);
   } }
```

In the above example, we created a **multidimensional array** named **a**. Since each component of a multidimensional array is similarly an array (**a[0]**, **a[1]** and **a[2]** are also arrays). Here, we are using the **length attribute** to calculate the length of each row.

Example: Print all the elements of 2d array Using Loop

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

```
int[ ][ ] a = {
   {1, -2, 3},
   {-4, -5, 6, 9},
   {7},
};
 for (int i = 0; i < a.length; ++i) {
   for(int j = 0; j < a[i].length; ++j) {
      System.out.println(a[i][j]);
   } } } }
```

We can also use the **for...each loop** to access elements of the multidimensional array. E.g.,

```
class MultidimensionalArray {
  public static void main(String[] args) {
    // create a 2d array
    int[ ][ ] a = {
       {1, -2, 3},
       {-4, -5, 6, 9},
       {7},
    };
   // first for...each loop access the individual array
    // inside the 2d array
     for (int[ ] innerArray: a) {
   // second for...each loop access each element inside the row
       for(int data: innerArray) {
          System.out.println(data);
       } } } }
```

In the above example, we created a 2d array named **a**. We then used for loop and for...each loop to access each element of the array.

HOW TO INITIALIZE A 3D ARRAY IN JAVA?

We can initialize a 3d array similar to the 2d array. For example,

```
// test is a 3d array
int[ ][ ][ ] test = {
      {
        {1, -2, 3},
        {2, 3, 4}
      },
      {
        {-4, -5, 6, 9},
```

```
        {1},
        {2, 3}
    } };
```

Basically, a **3d array** is an array of 2d arrays. The rows of a 3d array can also vary in length just like in a 2d array.

Example: 3-dimensional Array

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

        // create a 3d array
        int[ ][ ][ ] test = {
            {
              {1, -2, 3},
              {2, 3, 4}
            },
            {
              {-4, -5, 6, 9},
              {1},
              {2, 3}
            }};

        // for..each loop to iterate through elements of 3d array
        for (int[ ][ ] array2D: test) {
            for (int[ ] array1D: array2D) {
                for(int item: array1D) {
                    System.out.println(item);
                }}}}}
```

JAVA OBJECT ORIENTED PROGRAMMING

JAVA CLASS AND OBJECTS

Java is an object-oriented programming language. The main concept for the OOP approach is to break problems into smaller objects. An **object** is an 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 look at classes in Java.

JAVA CLASS

A **class** is a blueprint or template for an object. Before you create any **object**, you have to first define the **class**. We can think of a 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.

CREATING A CLASS IN JAVA

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

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

The **fields** (**variables**) and **methods** represent the object **state** and **behavior** respectively.

fields are used to store data

methods are used to perform some operations

For a 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");
  }}
```

We made a **Bicycle** class which contains **a field** named **gear** and **a method** named **braking**(). Here, a **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 a prototype. NB: We used keywords **private** and **public**. These are known as **access modifiers**.

JAVA OBJECTS

An **object** is a class instance e.g. **a Bicycle** is **a class** and **MountainBicycle**, **SportsBicycle**, **TouringBicycle**, etc can be considered as objects of the class.

CREATING AN OBJECT IN JAVA

```
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 a class to create an object. **Constructors** are the same as **methods** and have the **same name** as the **class name**. E.g., **Bicycle()** is a 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. NB: Fields and methods of a class are also called **members** of the class.

ACCESS MEMBERS OF A CLASS

We use the **name of objects** along with the **.operator** to access members of a class. E.g.,

```
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()
```

We have mentioned the word **method** a few times.

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 Main {
  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();
  }}
```

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 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 a 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.

E.g.: 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();
  }}
```

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

JAVA METHODS

A **method** is a block of code that performs a specific task. If 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.

USER-DEFINED METHODS.

Declaring a Java Method and the syntax to declare a method is:

```
returnType methodName() {
  // method body
}
```

Here,

- **returnType** - It specifies what type of value a method returns e.g. If a method has an int return type then it returns an integer value.
- If a **method does not return** a value, its return type is **void**.
- **methodName** - It is an identifier that is used to refer to the particular method in a program.
- **method body** - It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the curly braces **{ }.** For example,

```
int addNumbers() {
// code
}
```

In the above example, the name of the method is adddNumbers(). And, the return type is int.

This is a **simple syntax** of declaring a method. A **complete syntax** of declaring a method is

```
modifier static returnType nameOfMethod (parameter1, parameter2, ...) {
  // method body
}
```

Here,

- **modifier** - It defines access types whether the method is public, private, and so on.
- **static** - If we use the static keyword, it can be accessed without creating objects.

For example, the **sqrt() method** of standard **Math class** is static. Hence, we can directly call **Math.sqrt()** without creating an **instance** of Math class. **parameter1/parameter2** - These are values passed to a method. We can pass any number of arguments to a method.

CALLING A METHOD IN JAVA

In the above example, we declared a method named **addNumbers().** Now, to use the method, we need to **call it**. Here's is how we can call the addNumbers() method.

```
// calls the method
addNumbers();
```

we call a method in Java using the **name of the method** followed by **a parenthesis**

WORKING OF JAVA METHOD CALL

Example 1: Java Methods

```
class Main {
  // create a method
  public int addNumbers(int a, int b) {
    int sum = a + b;
    // return value
    return sum;
  }

  public static void main(String[ ] args) {
    int num1 = 25;
    int num2 = 15;

    // create an object of Main
    Main obj = new Main();
    // calling method
    int result = obj.addNumbers(num1, num2);
    System.out.println("Sum is: " + result);
  } }
```

In the above example, we have created a method named **addNumbers().** The method takes two parameters **a** and **b**. Notice the line,

```
int result = obj.addNumbers(num1, num2);
```

Here, we called **the method** by passing **two arguments num1** and **num2**. Since the method is returning some value, **we have stored the value in the result variable**. Note: The method is not static. Hence, we are calling the method using the object of the class.

JAVA METHOD RETURN TYPE

A **Java method** may or may not return a value to a function call. We use the **return statement** to return any value. For example,

```
int addNumbers() {
...
return sum;
}
```

Here, we are returning the variable **sum**. Since the return type of the function is **int**. The sum variable should be of int type. Otherwise, it will generate an error.

Example 2: Method Return Type

```
class Main {
// create a method
  public static int square(int num) {
    // return statement
    return num * num;
  }
  public static void main(String[ ] args) {
    int result;
    // call the method
    // store returned value to result
    result = square(10);
    System.out.println("Squared value of 10 is: " + result);
  } }
```

In the above program, we created **a method** named **square().** The method takes a number as its parameter and returns the square of the number. Here, we have mentioned the **return type** of the method as **int**. Hence, the method should always return an **integer value.**

Java method returns a value to the method call

NB: If a method does not **return any value**, we use the **void** keyword as the return type of the method. For example,

```
public void square(int a) {
  int square = a * a;
  System.out.println("Square is: " + square);
}
```

METHOD PARAMETERS IN JAVA

A **method parameter** is a value accepted by the method. As mentioned earlier, a method can also have any number of parameters. For example,

```
// method with two parameters
int addNumbers(int a, int b) {
  // code
}

// method with no parameter
int addNumbers(){
  // code
}
```

If a method is created with parameters, we need to pass the corresponding values while calling the method. For example,

```
// calling the method with two parameters
addNumbers(25, 15);

// calling the method with no parameters
addNumbers()
```

Example 3: Method Parameters

```
class Main {

// method with no parameter
  public void display1() {
    System.out.println("Method without parameter");
  }

 // method with single parameter
 public void display2(int a) {
   System.out.println("Method with a single parameter: " + a);
  }

  public static void main(String[ ] args) {

    // create an object of Main
    Main obj = new Main();

    // calling method with no parameter
    obj.display1();

    // calling method with the single parameter
    obj.display2(24);
  } }
```

METHOD WITHOUT PARAMETER

Method with a single parameter: 24

Here, the **parameter** of **the method** is **an int**. Hence, if we pass **any other data type** instead of int, **the compiler will throw an error**. It is because Java is **a strongly typed language.**

Note: The argument 24 passed to the **display2() method** during a **method call** is called the **actual argument.** The parameter **num** accepted by the **method definition** is known as **the formal argument**. We need to specify the **type of formal arguments**. And, the **type of actual arguments** and formal arguments should always match.

STANDARD LIBRARY METHODS

The **standard library methods** are built-in methods in Java that are readily available for use. Standard libraries come along with the **Java Class Library (JCL)** in a Java archive (***.jar**) file with **JVM** and **JRE**.

For example, **print()** is a method of java.io.PrintSteam. The print("...") method prints the string inside quotation marks. **sqrt()** is a method of Math class. It returns the square root of a number. Here's a working example:

Example 4: Java Standard Library Method

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

    // using the sqrt() method
    System.out.print("Square root of 4 is: " + Math.sqrt(4));
  }}
```

WHAT ARE THE ADVANTAGES OF USING METHODS?

1.The main benefit **is code reusability.** We can write a method once, and use it multiple times. We don't have to rewrite the whole code always, "**write once, reuse multiple times**".

Example 5: Java Method for Code Reusability

```
public class Main {
  // method defined
  private static int getSquare(int x){
    return x * x;
  }
  public static void main(String[ ] args) {
    for (int i = 1; i <= 5; i++) {
      // method call
      int result = getSquare(i);
      System.out.println("Square of " + i + " is: " + result);
    }}}
```

In the above program, we created the method named **getSquare()** to calculate the **square** of a number. Here, **a method** is used to calculate the square of numbers less than 6. So, the same method is used again and again.

2.Methods make code more readable and easier to debug. Here, the getSquare() method keeps the code to compute the square in a block. Hence, makes it more readable.

JAVA METHOD OVERLOADING

In Java, **two** or **more methods** may have the same name if they differ in parameters (different **number of parameters**, different **types of parameters**, or both). These methods are called **overloaded methods** and this feature is called method overloading. For example:

```
void func() { ... }
void func(int a) { ... }
float func(double a) { ... }
float func(int a, float b) { ... }
```

The **func() method** is overloaded. These methods have the same name but accept different arguments. NB: The **return types** of the above methods are not the same. It's because method overloading is not associated with return types. **Overloaded methods** may have the same or different return types, but they must differ in parameters.

WHY METHOD OVERLOADING? If, you have to add numbers with any number of arguments (let's say either **2** or **3** arguments for simplicity). In order to accomplish the task, you can create **two methods sum2num(int, int)** and **sum3num(int, int, int)** for two and three parameters respectively. However, other programmers, may get confused in the future as the behavior of both methods are the same but they differ by name.

A better way to achieve this task is by overloading methods. And, depending upon the argument passed, one of the overloaded methods is called. This helps to increase the readability of the program.

HOW TO PERFORM METHOD OVERLOADING IN JAVA?

1. Overloading by changing the number of parameters

```
class MethodOverloading {
   private static void display(int a){
      System.out.println("Arguments: " + a);
   }

   private static void display(int a, int b){
      System.out.println("Arguments: " + a + "and " + b);
   }
```

```
    public static void main(String[ ] args) {
        display(1);
        display(1, 4);
    }}
```

2. Method Overloading by changing the data type of parameters

```
class MethodOverloading {

    // this method accepts int
    private static void display(int a){
        System.out.println("Got Integer data.");
    }

    // this method  accepts String object
    private static void display(String a){
        System.out.println("Got String object.");
    }

    public static void main(String[] args) {
        display(1);
        display("Hello");
    }}
```

Here, both overloaded methods accept one argument. But, one accepts the argument of type int whereas other accepts String object.

Let's look at a real-world example:

```
class HelperService {

    private String formatNumber(int value) {
        return String.format("%d", value);
    }

    private String formatNumber(double value) {
        return String.format("%.3f", value);
    }

    private String formatNumber(String value) {
        return
String.format("%.2f", Double.parseDouble(value));
    }
public static void main(String[ ] args) {
        HelperService hs = new HelperService();
        System.out.println(hs.formatNumber(500));
```

```
        System.out.println(hs.formatNumber(89.9934));
        System.out.println(hs.formatNumber("550"));
    }}
```

Note: In Java, you can also overload constructors in a similar way like methods.

IMPORTANT POINTS

- Two or more methods can have the same name inside the same class if they accept different arguments. This feature is known as method overloading.
- Method overloading is achieved by either:
  changing the number of arguments.
  or changing the data type of arguments.
- It is not method overloading if we only change the return type of methods. There must be differences in the number of parameters.

JAVA CONSTRUCTORS

What is a Constructor? A **constructor** in Java is similar to a method that is invoked when an object of the class is created. Unlike Java methods, a constructor has the same name as that of the class and does not have any return type. For example,

```
class Test {
  Test() {
    // constructor body
  }}
```

Here, **Test()** is a constructor with the same name as that of the class and doesn't have a return type.

Example 1: Java Constructor

```
class Main {
  private String name;
  // constructor
  Main() {
    System.out.println("Constructor Called:");
    name = "ISBAT";
  }
  public static void main(String[ ] args) {
    // constructor is invoked while
    // creating an object of the Main class
    Main obj = new Main();
    System.out.println("The name is " + obj.name);
  }}
```

In the above example, we have created a constructor named Main(). Inside the constructor, we are initializing the value of the name variable.
Notice the statement of creating an object of the Main class.

```
Main obj = new Main();
```

Here, when an object is created, the **Main() constructor** is called. And, the value of the name variable is initialized. Hence, the program prints the value of the name variables as **ISBAT**.

TYPES OF CONSTRUCTOR

In Java, constructors can be divided into 3 types:

1. **No-Arg Constructor**
2. **Parameterized Constructor**
3. **Default Constructor**

1.**Java No-Arg Constructors:** Like **methods**, a Java constructor may or may not have any parameters (arguments). If a constructor does not accept any parameters, it is known as a no-argument constructor. For example,

```
private Constructor() {
  // body of the constructor
}
```

Example 2: Java private no-arg constructor

```
class Main {
  int i;
  // constructor with no parameter
  private Main() {
    i = 5;
    System.out.println("Constructor is called");
  }
  public static void main(String[ ] args) {
    // calling the constructor without any parameter
    Main obj = new Main();
    System.out.println("Value of i: " + obj.i);
  }}
```

In the above example, we created a **constructor Main()**. Here, the constructor does not accept any parameters. Hence, it is known as a no-arg constructor. Notice that we have declared the constructor as **private**. Once **a constructor** is declared **private**, it cannot be accessed from outside the class. So, creating objects outside a class is prohibited using the private constructor.

Here, we are creating the object inside the same class. Hence, the program is able to access the constructor. But, if you want to create objects outside the class, then you need to declare the **constructor** as **public**.

Example 3: Java public no-arg constructors

```
class Company {
  String name;
  // public constructor
  public Company() {
    name = "ISBAT";
  }}
class Main {
  public static void main(String[ ] args) {
    // object is created in another class
    Company obj = new Company();
    System.out.println("Company name = " + obj.name);
  }}
```

2. **Java Parameterized Constructor:** A Java constructor can similarly accept one or more parameters. Such constructors in Java are called **parameterized constructors** (constructor with parameters).

Example 4: Parameterized constructor

```
class Main {
  String languages;
  // constructor accepting single value
  Main(String lang) {
    languages = lang;
    System.out.println(languages + " Programming Language");
  }
  public static void main(String[] args) {
    // call constructor by passing a single value
    Main obj1 = new Main("Java");
    Main obj2 = new Main("Python");
    Main obj3 = new Main("C");
  }}
```

In the above example, we created a constructor named Main(). Here, the constructor takes a single parameter. Notice the expression,

```
Main obj1 = new Main("Java");
```

Here, we passed a single value to the constructor. Based on the argument passed, the language variable is initialized inside the constructor.

3. **Java Default Constructor:** If we don't create any constructor, Java compiler automatically creates a **no-arg constructor** during the execution of the program. This constructor is called a **default constructor.**

Example 5: Default Constructor

```
class Main {
  int a;
  boolean b;
  public static void main(String[] args) {
    // A default constructor is called
    Main obj = new Main();
    System.out.println("Default Value:");
    System.out.println("a = " + obj.a);
    System.out.println("b = " + obj.b);
  }}
```

Here, we haven't created any constructors. Hence, the Java compiler automatically creates the default constructor. The default constructor initializes any uninitialized instance variables with default values.

Type	Default Value
boolean	false
byte	0
short	0
int	0
long	0L
char	\u0000
float	0.0f
double	0.0d
object	Reference null

In the above program, variables **a** and **b** are initialized with a default value **0** and false respectively. The above program is equivalent to:

```
class Main {
  int a;
  boolean b;
  Main() {
    a = 0;
```

```
    b = false;
  }

  public static void main(String[ ] args) {
    // call the constructor
    Main obj = new Main();

    System.out.println("Default Value:");
    System.out.println("a = " + obj.a);
    System.out.println("b = " + obj.b);
  }}
```

IMPORTANT NOTES ON JAVA CONSTRUCTORS

- Constructors are invoked implicitly when you instantiate objects.
- The two rules for creating a constructor are:
- The name of the constructor should be the same as the class.
- A Java constructor must not have a return type.
- If a class doesn't have a constructor, the Java compiler automatically creates a default constructor during run-time. The default constructor initializes instance variables with default values. For example, the int variable will be initialized to 0

Constructor types:

- No-Arg Constructor - a constructor that does not accept any arguments
- Parameterized constructor - a constructor that accepts arguments
- Default Constructor - a constructor that is automatically created by the Java compiler if it is not explicitly defined.
- A constructor cannot be abstract or static or final.
- A constructor can be overloaded but cannot be overridden.

Constructors Overloading in Java

Similar to Java **method overloading**, we can also create two or more constructors with different parameters. This is called constructors overloading.

Example 6: Java Constructor Overloading

```
class Main {
  String language;
  // constructor with no parameter
  Main() {
    this.language = "Java";
  }
  // constructor with a single parameter
  Main(String language) {
    this.language = language;
```

```
}
public void getName() {
  System.out.println("Programming Langauage: " + this.language);
}
public static void main(String[] args) {
  // call constructor with no parameter
  Main obj1 = new Main();
  // call constructor with a single parameter
  Main obj2 = new Main("Python");
  obj1.getName();
  obj2.getName();
}}
```

In the above example, we have **two constructors**: **Main()** and **Main(String language).** Here, both the **constructor initialize** the value of the **variable language** with different values. Based on the parameter passed during object creation, different constructors are called and different values are assigned. It is also possible to call one constructor from another constructor. Note: We have used the **this keyword** to specify the variable of the class.

JAVA STRINGS

In Java, **a string** is a sequence of characters. For example, "**BULEGA**" is a string containing a sequence of characters '**B**', '**U**', '**L**', '**E**', '**G**', and '**A**'. We use **double quotes** to represent a string in Java. For example,

```
// CREATE A STRING
String type = "Java programming";
```

Here, we have created a string variable named type. The variable is initialized with the string Java Programming.

Example: Create a String in Java

```
class Main {
  public static void main(String[] args) {
    // create strings
    String first = "ISBAT";
    String second = "JAVA";
    String third = "Programming";
    // print strings
    System.out.println(first);   // print ISBAT
    System.out.println(second);  // print JAVA
    System.out.println(third);   // print Programming
  }}
```

In the above example, we created three strings named **first**, **second**, and **third**. Here, we are directly creating strings like primitive types. But, there is another way of creating Java strings (using the new keyword). NB: Strings in Java are not primitive types (like int, char, etc). Instead, all strings are objects of a predefined class named String. And, all string variables are instances of the **String class.**

JAVA STRING OPERATIONS

Java String provides various methods to perform different operations on strings.

1. Get length of a String

To find the length of a string, we use the **length() method** of the String. For example,

```
class Main {
  public static void main(String[ ] args) {
    // create a string
    String greet = "Hello! World";
    System.out.println("String: " + greet);
    // get the length of greet
    int length = greet.length();
    System.out.println("Length: " + length);
  }}
```

The **length() method** calculates the **total number** of characters in a string and returns it. To learn more, visit Java String length().

2. Join Two Java Strings

We can join two strings in Java using the concat() method. For example,

```
class Main {
  public static void main(String[] args) {
    // create first string
    String first = "Java ";
    System.out.println("First String: " + first);
    // create second
    String second = "Programming";
    System.out.println("Second String: " + second);
    // join two strings
    String joinedString = first.concat(second);
    System.out.println("Joined String: " + joinedString);
  }}
```

We have created two strings named **first** and **second**. Notice the statement,

```
String joinedString = first.concat(second);
```

Here, the **concat() method joins** the **second string** to the **first string** and assigns it to the **joinedString variable**. We can also join two strings using the + operator in Java. To learn more, visit Java String concat().

3. Compare two Strings

In Java, we can make **compare** between two strings using the **equals() method**. E.g.,

```
class Main {
  public static void main(String[ ] args) {
    // create 3 strings
    String first = "java programming";
    String second = "java programming";
    String third = "python programming";
    // compare first and second strings
    boolean result1 = first.equals(second);
    System.out.println("Strings first and second are equal: " + result1);
    // compare first and third strings
    boolean result2 = first.equals(third);
    System.out.println("Strings first and third are equal: " + result2);
  }}
```

In the above example, we have created **3 strings** named **first**, **second**, and **third**. Here, we use the **equal() method** to check if one string is equal to another. The **equals() method** checks the content of strings while comparing them. NB: We can also compare **two strings** using the == operator in Java. But, this approach is different than the **equals() method**. To learn more, visit Java String == vs equals().

Escape character in Java Strings

An escape character is used to escape some of the characters present inside a string. Suppose we need to include double quotes inside a string.

```
// include double quote
String example = "This is the "String" class";
```

Since **strings** are represented by **double quotes**, the compiler treats **"This is the "** as a string. Therefore, the above code will cause an error. To solve this issue, we use the **escape character ** in Java. For example,

```
// use the escape character
String example = "This is the \"String\" class.";
```

Now escape characters tell the compiler to escape double quotes and read the whole text.

Java Strings are Immutable

In Java, **strings are immutable.** This means, once we create a string, we cannot change that string. To understand it more deeply, consider an example:

```
// create a string
String example = "Hello! ";
```

Here, we have created a string variable named **example**. The variable holds astring **"Hello! ".** Now suppose you want to change the string.

```
// add another string "World"
```

```
// to the previous tring example
example = example.concat(" World");
```

Here, we are using the **concat() method** to add another string World to the previous string. It looks like we are able to change the value of the previous string. But, this is not true. Let's see what has happened here,

```
JVM takes the first string "Hello! "
creates a new string by adding "World" to the first string
assign the new string "Hello! World" to the example variable
the first string "Hello! " remains unchanged
Creating strings using the new keyword
```

SO FAR WE HAVE CREATED STRINGS LIKE PRIMITIVE TYPES IN JAVA.

Since strings in Java are objects, we can create strings using the **new keyword** as well. E.g.,

```
// create a string using the new keyword
String name = new String("Java String");
```

We create a string name using the **new keyword.** Here, when we create the **string object**, the **String() constructor** is invoked. NB: The **String class** provides various other constructors to create strings.

```
Example: Create Java Strings using the new keyword
class Main {
  public static void main(String[ ] args) {
    // create a string using new
    String name = new String("Java String");
    System.out.println(name);  // print Java String
  }}
```

Now that we know how strings are created using string literals and the new keyword, let's see what the major difference between them. In Java, the JVM maintains a string pool to store all its strings in memory. The string pool helps in reusing the strings.

1. While creating strings using string literals,

```
String example = "Java";
```

Here, we are directly providing the value of the string (Java). Hence, the compiler first checks the string pool to see if the string already exists. If the string already exists, the new string is not created. Instead, the new reference, example points to the already existed string (Java). If the string doesn't exist, the new string (Java is created.

2. While creating strings using the new keyword,

```
String example = new String("Java");
```

Here, the value of the string is not directly provided. Hence, a new "Java" string is created even though "Java" is already present inside the memory pool.

METHODS OF JAVA STRING

Besides those mentioned above, there are various **string methods** present in Java. Here are some of those methods:

METHODS	DESCRIPTION
contains()	checks whether the string contains a substring
substring()	returns the substring of the string
join()	join the given strings using the delimiter
replace()	replaces the specified old character with the specified new character
replaceAll()	replaces all substrings matching the regex pattern
replaceFirst()	replace the first matching substring
charAt()	returns the character present in the specified location
getBytes()	converts the string to an array of bytes
indexOf()	returns the position of the specified character in the string
compareTo()	compares two strings in the dictionary order
trim()	removes any leading and trailing whitespaces
format()	returns a formatted string
split()	breaks the string into an array of strings
toLowerCase()	converts the string to lowercase
toUpperCase()	converts the string to uppercase
valueOf()	returns the string representation of the specified argument
toCharArray()	converts the string to a char array
matches()	checks whether the string matches the given regex
startsWith()	checks if the string begins with the given string
endsWith()	checks if the string ends with the given string
isEmpty()	checks whether a string is empty of not
intern()	returns the canonical representation of the string
contentEquals()	checks whether the string is equal to charSequence
hashCode()	returns a hash code for the string
subSequence()	returns a subsequence from the string
compareToIgnoreCase()	compares two strings ignoring case differences

JAVA ACCESS MODIFIERS

In Java, **access modifiers** are used to set the accessibility (visibility) of classes, interfaces, variables, methods, constructors, data members, and the setter methods. For example,

```
class Animal {
  public void method1() {...}
  private void method2() {...}
}
```

In the above example, we have declared 2 methods: method1() and method2(). Here,

method1 is public - This means it can be accessed by other classes.
method2 is private - This means it cannot be accessed by other classes.

Note the keywords **public** and **private**. These are **access modifiers**. They are also known as **visibility modifiers**. Note: You cannot set the access modifier of getters methods.

TYPES OF ACCESS MODIFIER

Before you learn about types of access modifiers, make sure you know about Java Packages. There are **four access modifiers keywords** in Java and they are:

MODIFIER	DESCRIPTION
1.Default	declarations are visible only within the package (package private)
2.Private	declarations are visible within the class only
3.Protected	declarations are visible within the package or all subclasses
4.Public	declarations are visible everywhere

Default Access Modifier

If we do not explicitly specify any access modifier for classes, methods, variables, etc, then by default the default access modifier is considered. For example,

```
package defaultPackage;
class Logger {
   void message(){
      System.out.println("This is a message");
   }}
```

Here, the **Logger class** has the default access modifier and the class is visible to all the classes that belong to the **defaultPackage package**. But, if we try to use the **Logger class** in another class outside of defaultPackage, you will get a **compilation error.**

Private Access Modifier

When **variables** and **methods** are declared **private**, they cannot be accessed outside of the class. For example,

```
class Data {
   // private variable
   private String name;
}
public class Main {
   public static void main(String[ ] main){
      // create an object of Data
      Data d = new Data();
```

```
        // access private variable and field from another class
        d.name = "ISBAT";
    }}
```

In the above example, we have declared a **private variable** named **name.** When we run the program, we will get the following error:

```
Main.java:18: error: name has private access in Data
    d.name = "ISBAT";
     ^
```

The error is generated because we are trying to access the **private variable** of the Data class from the Main class. You might be wondering what if we need to access those private variables. In this case, we can use the **getters** and **setters methods**. For example,

```
class Data {
    private String name;
    // getter method
    public String getName() {
        return this.name;
    }
    // setter method
    public void setName(String name) {
        this.name= name;
    }}
public class Main {
    public static void main(String[ ] main){
        Data d = new Data();
       // access the private variable using the getter and setter
        d.setName("ISBAT");
        System.out.println(d.getName());
    }}
```

In the above example, we have a **private variable** named **name**. In order to access the variable from the outer class, we used **methods**: **getName()** and **setName().** These methods are called **getter** and **setter** in Java. Here, we used a **setter method (setName())** to assign a value to the variable and a **getter method (getName())** to access a variable. We have also used the **this keyword** inside the **setName()** to refer to the variable of the class.

NB: We can't declare classes and interfaces private in Java. However, the nested classes can be declared private.

Protected Access Modifier

When methods and data members are declared protected, we can access them within the same package as well as from subclasses. For example,

```
class Animal {
   // protected method
    protected void display() {
       System.out.println("I am an animal");
    }}
class Dog extends Animal {
    public static void main(String[ ] args) {
      // create an object of Dog class
       Dog dog = new Dog();
       // access protected method
       dog.display();
    }}
```

We have **a protected method** named **display()** inside the **Animal class**. The **Animal class** is inherited by the **Dog class**. We then created an object dog of the Dog class. Using the object we tried to access the protected method of the parent class. Since protected methods can be accessed from the child classes, we are able to access the method of Animal class from the Dog class. Note: We cannot declare classes or interfaces protected in Java.

Public Access Modifier

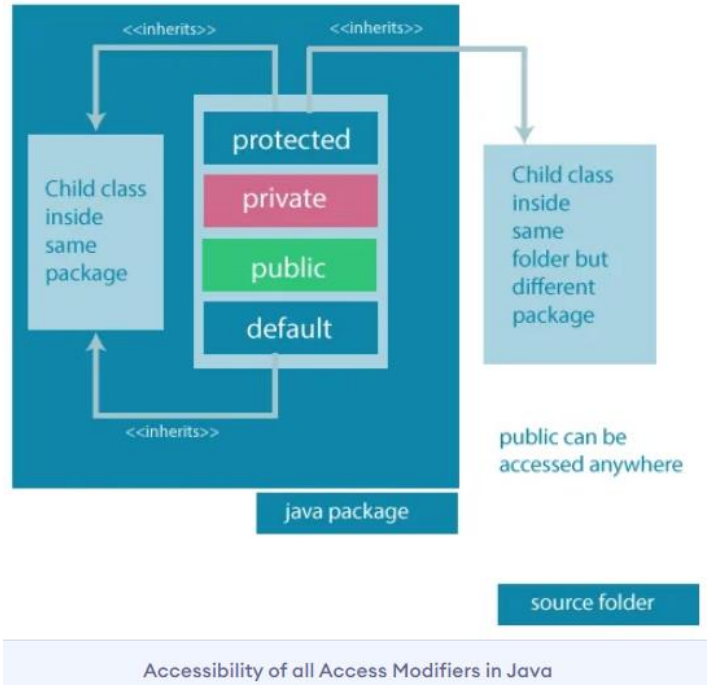
When methods, variables, classes, and so on are declared public, then we can access them from anywhere. The public access modifier has no scope restriction. For example,

```
// Animal.java file
// public class
public class Animal {
   // public variable
    public int legCount;
   // public method
    public void display() {
       System.out.println("I am an animal.");
       System.out.println("I have " + legCount + " legs.");
    }}
// Main.java
public class Main {
    public static void main( String[] args ) {
      // accessing the public class
       Animal animal = new Animal();
      // accessing the public variable
       animal.legCount = 4;
      // accessing the public method
```

```
        animal.display();
    }}
```

- The public class Animal is accessed from the Main class.
- The public variable legCount is accessed from the Main class.
- The public method display() is accessed from the Main class.

Access Modifiers Summarized in one figure



Accessibility of all Access Modifiers in Java

Access modifiers are mainly used for encapsulation. It help us to control what part of a program can access the members of a class. So that misuse of data can be prevented.

JAVA THIS KEYWORD

In Java, the **this keyword** is used to refer to the current object inside a method or a constructor. E.g.,

```
class Main {
    int instVar;
    Main(int instVar){
        this.instVar = instVar;
        System.out.println("this reference = " + this);
    }
    public static void main(String[ ] args) {
        Main obj = new Main(8);
        System.out.println("object reference = " + obj);
    }}
```

We created an object named **obj** of the class Main. We then print the reference to the object **obj** and this keyword of the class. Here, we can see that the **reference** of both **obj** and **this** is the same. It means that the this keyword is nothing but a reference to the current object.

USE OF THE this Keyword

There are various situations where **this keyword** is commonly used.

Using this for Ambiguity Variable Names

In Java, it is not allowed to declare two or more variables having the same name inside a scope (class scope or method scope). But, instance variables and parameters may have the same name. For example,

```
class MyClass {
   // instance variable
   int age;
   // parameter
   MyClass(int age){
      age = age;
   }}
```

In the above program, the instance variable and the parameter have the same name: age. Here, the Java compiler is confused due to name ambiguity. In such a situation, we use this keyword. For example, First, let's see an example without using this keyword:

```
class Main {
   int age;
   Main(int age){
      age = age;
   }
   public static void main(String[ ] args) {
      Main obj = new Main(8);
      System.out.println("obj.age = " + obj.age);
   }}
```

We have passed **8** as a value to the constructor. However, we are getting **0** as an output. This is because the Java compiler gets confused because of the ambiguity in names between instance the variable and the parameter. Now, let's rewrite the above code using this keyword.

```
class Main {
   int age;
   Main(int age){
      this.age = age;
   }
   public static void main(String[] args) {
```

```
        Main obj = new Main(8);
        System.out.println("obj.age = " + obj.age);
    }}
```

Now, we are getting the expected output. It is because when the constructor is called, this inside the constructor is replaced by an **object obj** that has called the constructor. Hence the age variable is assigned value 8. Similarly, if the name of the parameter and instance variable is different, the compiler automatically appends this keyword. For example, the code:

```
class Main {
    int age;
    Main(int i) {
        age = i;
    }}
```

is equivalent to:

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

Another common use of the **this keyword** is in setters and getters methods of a class. E.g.,

```
class Main {
    String name;
    // setter method
    void setName( String name ) {
        this.name = name;
    }
    // getter method
    String getName(){
        return this.name;
    }
    public static void main( String[] args ) {
        Main obj = new Main();
        // calling the setter and the getter method
        obj.setName("Toshiba");
        System.out.println("obj.name: "+obj.getName());
    }}
```

Here, we have used this keyword:

- to assign value inside the setter method
- to access value inside the getter method

Using this in Constructor Overloading

When working with **constructor overloading**, you may have to invoke **one constructor** from **another constructor**. In such a case, we cannot call the constructor explicitly. Instead, we have to use **this keyword**. Here, we use a different form of this keyword. That is, this(). E.g.

```
class Complex {
  private int a, b;
  // constructor with 2 parameters
  private Complex( int i, int j ){
    this.a = i;
    this.b = j;
  }
  // constructor with single parameter
  private Complex(int i){
    // invokes the constructor with 2 parameters
    this(i, i);
  }
  // constructor with no parameter
  private Complex(){
    // invokes the constructor with single parameter
    this(0);
  }
  @Override
  public String toString(){
    return this.a + " + " + this.b + "i";
  }
  public static void main( String[] args ) {
    // creating object of Complex class
    // calls the constructor with 2 parameters
    Complex c1 = new Complex(2, 3);
    // calls the constructor with a single parameter
    Complex c2 = new Complex(3);
    // calls the constructor with no parameters
    Complex c3 = new Complex();
    // print objects
    System.out.println(c1);
```

```
        System.out.println(c2);
        System.out.println(c3);
    }}
```

In the above example, we have used this keyword,

- to call the constructor Complex(int i, int j) from the constructor Complex(int i)
- to call the constructor Complex(int i) from the constructor Complex()

Notice the line,

```
System.out.println(c1);
```

Here, when we print the **object c1**, the object is converted into a string. In this process, the **toString()** is called. Since we override the toString() method inside our class, we get the output according to that method.

One of the huge advantages of **this()** is to reduce the amount of duplicate code. But, we should be always careful while using the **this()**. This is because **calling a constructor** from another **constructor** adds overhead and it is a slow process. Another huge advantage of using this() is to reduce the amount of duplicate code. Note: Invoking one constructor from another constructor is called **explicit constructor invocation.**

Passing this as an Argument

We can use this keyword to pass the current object as an argument to a method. For example,

```
class ThisExample {
  // declare variables
  int x;
  int y;

  ThisExample(int x, int y) {
    // assign values of variables inside constructor
    this.x = x;
    this.y = y;

    // value of x and y before calling add()
    System.out.println("Before passing this to addTwo() method:");
    System.out.println("x = " + this.x + ", y = " + this.y);

    // call the add() method passing this as argument
    add(this);

    // value of x and y after calling add()
    System.out.println("After passing this to addTwo() method:");
    System.out.println("x = " + this.x + ", y = " + this.y);
  }

  void add(ThisExample o){
```

```
        o.x += 2;
        o.y += 2;
    }}
class Main {
    public static void main( String[ ] args ) {
        ThisExample obj = new ThisExample(1, -2);
    }}
```

In the above example, inside the constructor ThisExample(), notice the line,

```
add(this);
```

Here, we called the **add() method** by passing **this** as an argument. Since this keyword contains the reference to the **object obj** of the class, we can change the value of **x** and **y** inside the add() method.

JAVA FINAL KEYWORD

In Java, the final keyword is used to denote constants. It can be used with variables, methods, and classes. If any entity (variable, method or class) is declared final, it can only be assigned once. That is,

- the final variable cannot be reinitialized with another value
- the final method cannot be overridden
- the final class cannot be extended

1. Java final Variable

In Java, we cannot change the value of a final variable. For example,

```
class Main {
  public static void main(String[ ] args) {
    // create a final variable
    final int AGE = 32;
    // try to change the final variable
    AGE = 45;
    System.out.println("Age: " + AGE);
  }}
```

We have created a final variable named age. And we have tried to change the value of the final variable. When we run the program, we will get a compilation error with the following message.

```
You cannot assign a value to final variable AGE
        AGE = 45;
        ^
```

Note: It is recommended to use uppercase to declare final variables in Java.

2. Java final Method

In Java, the final method cannot be overridden by the child class. For example,

```
class FinalDemo {
   // create a final method
    public final void display() {
      System.out.println("This is a final method.");
    }}
class Main extends FinalDemo {
  // try to override final method
  public final void display() {
    System.out.println("The final method is overridden.");
  }
  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }}
```

We have created a final method named **display()** inside the **FinalDemo class**. Here, the **Main class inherits** the **FinalDemo class**. We tried to override the final method in the Main class. When we run the program, we will get a compilation error with the following message.

```
display() in Main cannot override display() in FinalDemo
 public final void display() {
                   ^
 overridden method is final
```

3. Java final Class

In Java, the final class cannot be inherited by another class. For example,

```
// create a final class
final class FinalClass {
  public void display() {
    System.out.println("This is a final method.");
  }}
// try to extend the final class
class Main extends FinalClass {
  public  void display() {
    System.out.println("The final method is overridden.");
  }
 public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
```

```
}}
```

In the above example, we have created a final class named FinalClass. Here, we have tried to inherit the final class by the Main class. When we run the program, we will get a compilation error with the following message.

```
cannot inherit from final FinalClass
class Main extends FinalClass {
```

JAVA RECURSION

In Java, a method that calls itself is known as a **recursive method**. And, this process is known as **recursion**. A physical example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

How Recursion works?

- A function is calling itself

Working of Java Recursion

We have called the **recurse() method** from inside the main method. (normal method call). And, inside the **recurse() method,** we are again calling the same recurse method. This is a recursive call. In order to stop the recursive call, we need to provide some conditions inside the method. Otherwise, the method will be called infinitely. Hence, we use the if...else statement (or similar approach) to terminate the recursive call inside the method.

Example: Factorial of a Number Using Recursion

```
class Factorial {
   static int factorial( int n ) {
      if (n != 0)  // termination condition
         return n * factorial(n-1); // recursive call
      else
         return 1;
   }
   public static void main(String[ ] args) {
      int number = 4, result;
      result = factorial(number);
      System.out.println(number + " factorial = " + result);
   }}
```

We have a method named **factorial().** The factorial() is called from the main() method. with the number variable passed as an argument. Here, notice the statement,

```
return n * factorial(n-1);
```

The **factorial() method** is calling itself. Initially, the value of **n** is **4** inside factorial(). During the next **recursive call**, **3** is passed to the **factorial() method.** This process continues until **n** is equal to **0.** When **n** is equal to **0**, the **if statement returns false** hence 1 is returned. Finally, the accumulated result is passed to the main() method.

Advantages and Disadvantages of Recursion

When a recursive call is made, new storage locations for variables are allocated on the stack. As, each recursive call returns, the old variables and parameters are removed from the stack. Hence, recursion generally uses more memory and is generally slow. On the other hand, a recursive solution is much simpler and takes less time to write, debug and maintain.
Recommended Reading: What are the advantages and disadvantages of recursion?

JAVA INSTANCEOF OPERATOR

instanceof operator in Java is used to check whether an object is an instance of a particular class or not. Its syntax is

objectName instanceOf className;

Here, if **objectName** is an instance of **className**, the operator returns true. Else, it returns false.

Example: Java instanceof

```
class Main {
  public static void main(String[ ] args) {
    // create a variable of string type
    String name = "Programiz";
    // checks if name is instance of String
    boolean result1 = name instanceof String;
    System.out.println("name is an instance of String: " + result1);
    // create an object of Main
    Main obj = new Main();
   // checks if obj is an instance of Main
    boolean result2 = obj instanceof Main;
    System.out.println("obj is an instance of Main: " + result2);
  }}
```

We have created a variable name of the String type and an object obj of the Main class. Here, we have used the **instanceof operator** to check whether name and obj are instances of the String and Main class respectively. And, the operator returns true in both cases. Note: In Java, String is a class rather than a primitive data type. To learn more, visit Java String.

Java instanceof during Inheritance

We can use the instanceof operator to check if objects of the subclass is also an instance of the superclass. For example,

```
// Java Program to check if an object of the subclass
// is also an instance of the superclass
// superclass
class Animal {
}
// subclass
class Dog extends Animal {
}
class Main {
  public static void main(String[ ] args) {
    // create an object of the subclass
    Dog d1 = new Dog();
    // checks if d1 is an instance of the subclass
    System.out.println(d1 instanceof Dog);       // prints true
    // checks if d1 is an instance of the superclass
    System.out.println(d1 instanceof Animal);    // prints true
  }}
```

We have created a subclass Dog that inherits from the superclass Animal. We have created an object d1 of the Dog class. Inside the print statement, notice the expression,

```
d1 instanceof Animal
```

Here, we are using the instanceof operator to check whether d1 is also an instance of the superclass Animal.

JAVA INHERITANCE

Inheritance is one of the key **features of OOP** that allows us to create a new class from an existing class. The **new class** that is created is known as subclass (child or derived class) and the existing class from where the child class is derived is known as superclass (parent or base class). The **extends keyword** is used to perform inheritance in Java. For example,

```
class Animal {
  // methods and fields
}
// use of extends keyword
// to perform inheritance
class Dog extends Animal {

  // methods and fields of Animal
  // methods and fields of Dog
```

```
}
```

The Dog class is created by inheriting the methods and fields from the Animal class. Here, Dog is the subclass and Animal is the superclass.

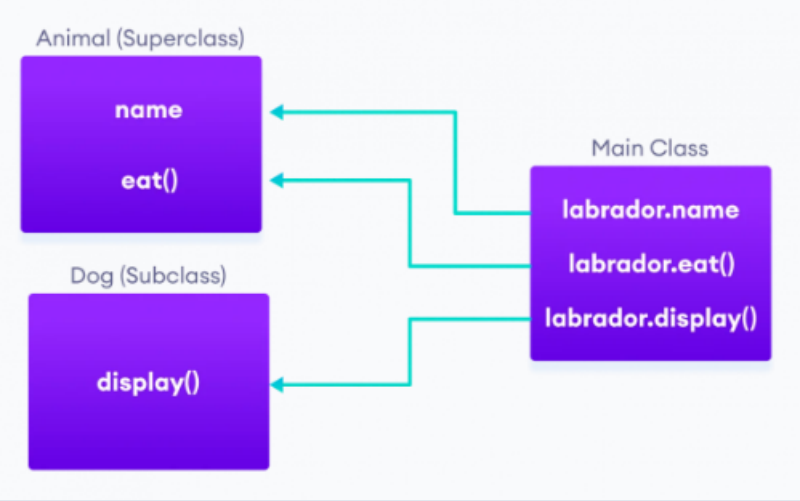
Example 1: Java Inheritance

```
class Animal {
  // field and method of the parent class
  String name;
  public void eat() {
    System.out.println("I can eat");
  }}
// inherit from Animal
class Dog extends Animal {
// new method in subclass
  public void display() {
    System.out.println("My name is " + name);
  }}
class Main {
  public static void main(String[] args) {
    // create an object of the subclass
    Dog labrador = new Dog();
    // access field of superclass
    labrador.name = "Rohu";
    labrador.display();
    // call method of superclass
    // using object of subclass
    labrador.eat();
  }}
```

We have derived a subclass Dog from superclass Animal. Notice the statements,

```
labrador.name = "Rohu";
labrador.eat();
```

Here, **labrador** is an object of Dog. But, name and **eat()** are the members of the Animal class. Since Dog inherits the field and method from Animal, we are able to access the field and method using the object of the Dog.



Subclass Dog can access the field and method of the superclass Animal.

Java Inheritance Implementation

In Java,we use inheritance only if there exists a relationship between two classes. For example,

Car is a Vehicle
Orange is a Fruit
Surgeon is a Doctor
Dog is an Animal

Here, Car can inherit from Vehicle, Orange can inherit from Fruit, and so on.

Method Overriding in Java Inheritance

In Example 1, we see the object of the subclass can access the method of the superclass. But, if the same method is present in both the superclass and subclass, what will happen? In this case, the method in the subclass overrides the method in the superclass. This concept is known as **method overriding** in Java.

Example 2: Method overriding in Java Inheritance

```
class Animal {
  // method in the superclass
  public void eat() {
    System.out.println("I can eat");
  }}
// Dog inherits Animal
class Dog extends Animal {
  // overriding the eat() method
  @Override
  public void eat() {
    System.out.println("I eat dog food");
  }
  // new method in subclass
```

```
  public void bark() {
    System.out.println("I can bark");
  }}
class Main {
  public static void main(String[ ] args) {
    // create an object of the subclass
    Dog labrador = new Dog();
    // call the eat() method
    labrador.eat();
    labrador.bark();
```

The eat() method is present in both the superclass Animal and the subclass Dog. Here, we have created an object labrador of Dog. Now when we call eat() using the object labrador, the method inside Dog is called. This is because the method inside the derived class overrides the method inside the base class. This is called method overriding. Note: We have used the @Override annotation to tell the compiler that we are overriding a method. However, the annotation is not mandatory. To learn more, visit Java Annotations.

super Keyword in Java Inheritance

Previously we saw that the same method in the subclass overrides the method in superclass. In such a situation, the super keyword is used to call the method of the parent class from the method of the child class.

Example 3: super Keyword in Inheritance

```
class Animal {
  // method in the superclass
  public void eat() {
    System.out.println("I can eat");
  }}
// Dog inherits Animal
class Dog extends Animal {
  // overriding the eat() method
  @Override
  public void eat() {
    // call method of superclass
    super.eat();
    System.out.println("I eat dog food");
  }
```

```
  // new method in subclass
  public void bark() {
    System.out.println("I can bark");
  }}
class Main {
  public static void main(String[ ] args) {
    // create an object of the subclass
    Dog labrador = new Dog();
    // call the eat() method
    labrador.eat();
    labrador.bark();
  }}
```

the eat() method is present in both the base class Animal and the derived class Dog. Notice the statement,

```
super.eat();
```

Here, the super keyword is used to call the eat() method present in the superclass. We can also use the super keyword to call the constructor of the superclass from the constructor of the subclass.

protected Members in Inheritance

In Java, if a class includes protected fields and methods, then these fields and methods are accessible from the subclass of the class.

Example 4: protected Members in Inheritance

```
class Animal {
  protected String name;
  protected void display() {
    System.out.println("I am an animal.");
  }}
class Dog extends Animal {
  public void getInfo() {
    System.out.println("My name is " + name);
  }}
class Main {
  public static void main(String[ ] args) {
    // create an object of the subclass
    Dog labrador = new Dog();
    // access protected field and method
```

```
        // using the object of subclass
        labrador.name = "Rocky";
        labrador.display();
        labrador.getInfo();
    }}
```

We created a class named Animal. The class includes a protected field: name and a method: display(). We have inherited the Dog class inherits Animal. Notice the statement,

```
labrador.name = "Rocky";
labrador.display();
```

Here, we are able to access the protected field and method of the superclass using the labrador object of the subclass.

Why use inheritance?

- The most important use of inheritance in Java is code reusability. The code that is present in the parent class can be directly used by the child class.
- Method overriding is also known as runtime polymorphism. Hence, we can achieve Polymorphism in Java with the help of inheritance.

TYPES OF INHERITANCE

There are five types of inheritance.

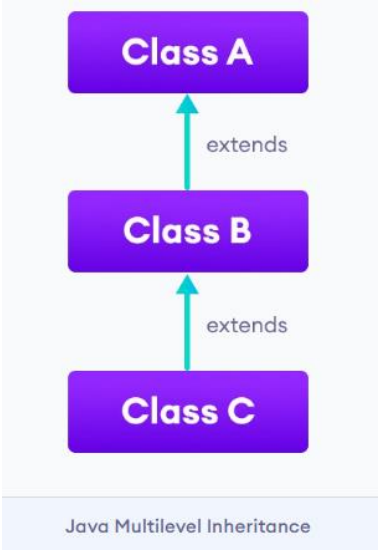
1. Single Inheritance

In single inheritance, a single subclass extends from a single superclass. For example,

Class A inherits from class B.
Java Single Inheritance

2. Multilevel Inheritance

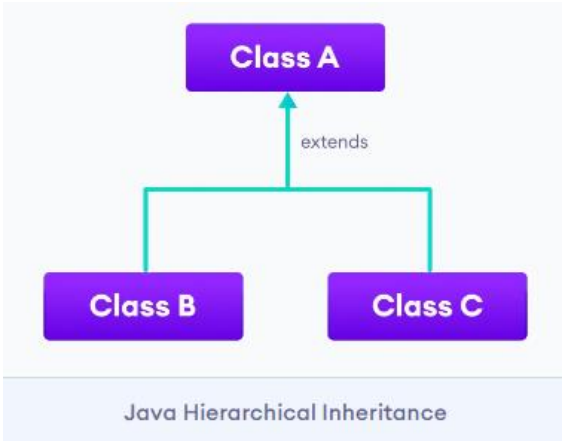
In multilevel inheritance, a subclass extends from a superclass and then the same subclass acts as a superclass for another class. For example,



Class B inherits from class A and class C inherits from class B.
Java Multilevel Inheritance

3. Hierarchical Inheritance

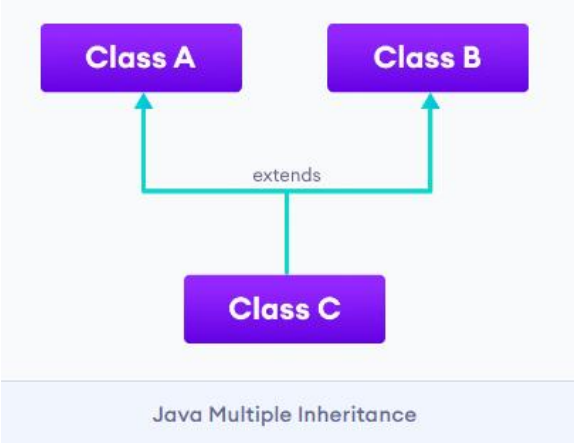
In hierarchical inheritance, multiple subclasses extend from a single superclass. For example,



Both classes B and C inherit from the single class A.
Java Hierarchical Inheritance

4. Multiple Inheritance

In multiple inheritance, a single subclass extends from multiple superclasses. For example,

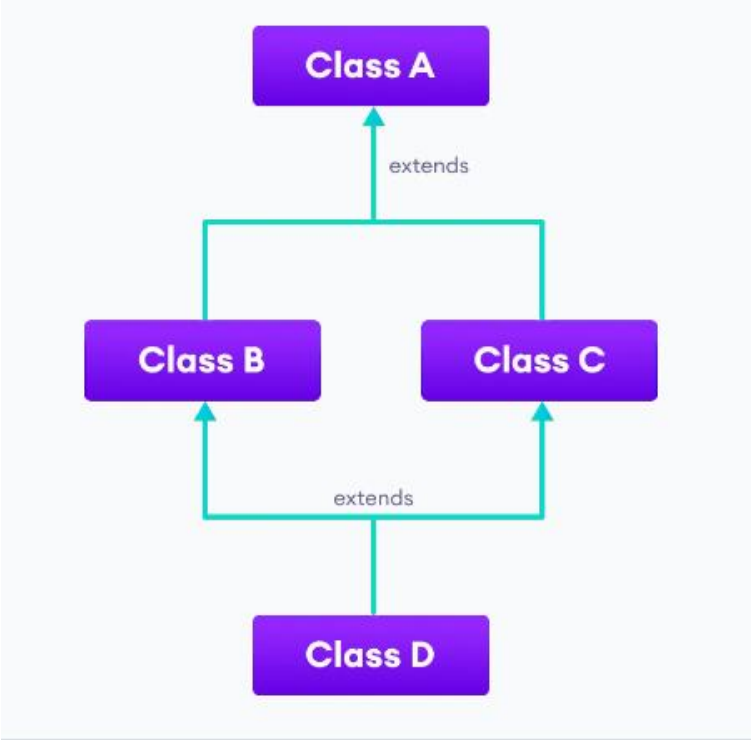


Class C inherits from both classes A and B.
Java Multiple Inheritance

Note: Java doesn't support multiple inheritance. However, we can achieve multiple inheritance using interfaces. To learn more, visit Java implements multiple inheritance.

5. Hybrid Inheritance

Hybrid inheritance is a combination of two or more types of inheritance. For example,



Java Hybrid Inheritance

Class B and C inherit from a single class A and class D inherits from both the class B and C.

Java Hybrid Inheritance

Here, we have combined hierarchical and multiple inheritance to form a hybrid inheritance.

JAVA METHOD OVERRIDING

Inheritance is an OOP property that allows us to derive a new class (subclass) from an existing class (superclass). The subclass inherits the attributes and methods of the superclass. Now, if the same method is defined in both the superclass and the subclass, then the method of the subclass class overrides the method of the superclass. This is known as method overriding.

Example 1: Method Overriding

```
class Animal {
  public void displayInfo() {
    System.out.println("I am an animal.");
  }}

class Dog extends Animal {
  @Override
  public void displayInfo() {
```

```
    System.out.println("I am a dog.");
  }}
class Main {
  public static void main(String[] args) {
    Dog d1 = new Dog();
    d1.displayInfo();
  }}
```

In the above program, the displayInfo() method is present in both the Animal superclass and the Dog subclass. When we call the **displayInfo()** using the d1 object (object of the subclass), the method inside the subclass Dog is called. The displayInfo() method of the subclass overrides the same method of the superclass.

Working of method overriding in Java.

Notice the use of **@Override** annotation in our example. In Java, annotations are the metadata that we used to provide information to the compiler. Here, the @Override annotation specifies the compiler that the method after this annotation overrides the method of the superclass.

It is not mandatory to use @Override. However, when we use this, the method should follow all the rules of overriding. Otherwise, the compiler will generate an error.

Java Overriding Rules

Both the superclass and the subclass must have the same method name, the same return type and the same parameter list.

- We cannot override the method declared as final and static.
- We should always override abstract methods of the superclass.

super Keyword in Java Overriding

A common question that arises while performing overriding in Java is: Can we access the method of the superclass after overriding? Well, the answer is Yes. To access the method of the superclass from the subclass, we use the super keyword.

Example 2: Use of super Keyword

```
class Animal {
  public void displayInfo() {
    System.out.println("I am an animal.");
  }}
class Dog extends Animal {
  public void displayInfo() {
    super.displayInfo();
    System.out.println("I am a dog.");
  } }
```

```
class Main {
  public static void main(String[ ] args) {
    Dog d1 = new Dog();
    d1.displayInfo();
  } }
```

The subclass **Dog** overrides the method displayInfo() of the superclass Animal. When we call the method displayInfo() using the d1 object of the Dog subclass, the method inside the Dog subclass is called; the method inside the superclass is not called.

Inside displayInfo() of the Dog subclass, we have used super.displayInfo() to call displayInfo() of the superclass. It is important to note that constructors in Java are not inherited. Hence, there is no such thing as **constructor overriding** in Java. But, we can call the constructor of the superclass from its subclasses. For that, we use super().

Access Specifiers in Method Overriding

Same method declared in a superclass and its subclasses can have different access specifiers. But, there is a restriction. We can only use those access specifiers in subclasses that provide larger access than the access specifier of the superclass. For example, Suppose, a method myClass() in the superclass is declared protected. Then, the same method myClass() in the subclass can be either public or protected, but not private.

Example 3: Access Specifier in Overriding

```
class Animal {
  protected void displayInfo() {
    System.out.println("I am an animal.");
  }}

class Dog extends Animal {
  public void displayInfo() {
    System.out.println("I am a dog.");
  }}

class Main {
  public static void main(String[] args) {
    Dog d1 = new Dog();
    d1.displayInfo();
  }}
```

The subclass Dog overrides the method displayInfo() of the superclass Animal. Whenever we call displayInfo() using the d1 (object of the subclass), the method inside the subclass is called.

Notice that, the displayInfo() is declared protected in the Animal superclass. The same method has the public access specifier in the Dog subclass. This is possible because the public provides larger access than the protected.

Overriding Abstract Methods

In Java, abstract classes are created to be the superclass of other classes. And, if a class contains an abstract method, it is mandatory to override it.