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. To learn more, visit Java 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()`

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. It has the same name as that of the class and doesn't have a return type.
Recommended Reading: Why do constructors not return values

Example 1: Java Constructor

```
class Main {
  private String name;

  // constructor
  Main() {
    System.out.println("Constructor Called:");
    name = "Programiz";
```

```
  }

  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);
  }
}
```

Output:

```
Constructor Called:
The name is Programiz
```

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 the 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 `Programiz`.

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

Similar to 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);
  }
}
```

Output:

```
Constructor is called
Value of i: 5
```

In the above example, we have 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 from

outside the 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. To learn more, visit Java Implement Private Constructor.
However, if we want to create objects outside the class, then we need to declare the constructor as `public`.
Example 3: Java public no-arg constructors

```
class Company {
  String name;

  // public constructor
  public Company() {
    name = "Programiz";
  }
}

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);
  }
}
```

Output:

```
Company name = Programiz
```

Recommended Reading: Java Access Modifier

---

2. Java Parameterized Constructor

A Java constructor can also accept one or more parameters. Such constructors are known as 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");
  }
}
```

Output:

```
Java Programming Language
Python Programming Language
C Programming Language
```

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

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

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

---

3. Java Default Constructor

If we do not create any constructor, the Java compiler automatically create a no-arg constructor during the execution of the program. This constructor is called 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);
  }
}
```

Output:

```
a = 0
b = false
```

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

```
class Main {
```

```
  int a;
  boolean b;

  // a private constructor
  private 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);
  }
}
```

The output of the program is the same as Example 5.

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 can not 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();
  }
}
```

Output:

```
Programming Language: Java
Programming Language: Python
```

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

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) { ... }
```

Here, the `func()` method is overloaded. These methods have the same name but accept different arguments.

> **Note**: The return types of the above methods are not the same. It is 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?
Suppose, you have to perform the addition of given numbers but there can be 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, as well as you in the future may get confused as the behavior of both methods are the same but they differ by name.
The better way to accomplish 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?

Here are different ways to perform method overloading:

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);
    }
}
```

Output:

```
Arguments: 1
```

```
Arguments: 1 and 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");
    }
}
```

Output:

```
Got Integer data.
Got String object.
```

Here, both overloaded methods accept one argument. However, 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"));
    }
}
```

When you run the program, the output will be:

```
500
89.993
550.00
```

> **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

What is a static keyword in Java?

In Java, if we want to access class members, we must first create an instance of the class. But there will be situations where we want to access class members without creating any variables.

In those situations, we can use the `static` keyword in Java. If we want to access class members without creating an instance of the class, we need to declare the class members static.
The `Math` class in Java has almost all of its members static. So, we can access its members without creating instances of the Math class. For example,

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

        // accessing the methods of the Math
class
```

```
        System.out.println("Absolute value of -
12 =  " + Math.abs(-12));
        System.out.println("Value of PI = " +
Math.PI);
        System.out.println("Value of E = " +
Math.E);
        System.out.println("2^2 = " +
Math.pow(2,2));
    }
}
```

Output:

```
Absolute value of -12 = 12
Value of PI = 3.141592653589793
Value of E = 2.718281828459045
2^2 = 4.0
```

In the above example, we have not created any instances of the `Math`class. But we are able to access its methods: `abs()` and `pow()` and variables: `PI` and `E`.
It is possible because the methods and variables of the `Math` class are static.

Static Methods

Static methods are also called class methods. It is because a static method belongs to the class rather than the object of a class.

And we can invoke static methods directly using the class name. For example,

```
class Test {
    // static method inside the Test class
    public static void method() {...}
}

class Main {
    // invoking the static method
    Test.method();
}
```

Here, we can see that the static method can be accessed directly from other classes using the class name.

In every Java program, we have declared the `main` method `static`. It is because to run the program the JVM should be able to invoke the main method during the initial phase where no objects exist in the memory.
Example 1: Java static and non-static Methods

```
class StaticTest {

    // non-static method
```

```
    int multiply(int a, int b){
        return a * b;
    }

    // static method
    static int add(int a, int b){
        return a + b;
    }
}

public class Main {

   public static void main( String[] args ) {

        // create an instance of the StaticTest
class
        StaticTest st = new StaticTest();

        // call the nonstatic method
        System.out.println(" 2 * 2 = " +
st.multiply(2,2));

        // call the static method
        System.out.println(" 2 + 3 = " +
StaticTest.add(2,3));
   }
}
```

Output:

```
2 * 2 = 4
```

```
2 + 3 = 5
```

In the above program, we have declared a non-static method named `multiply()` and a static method named `add()` inside the class `StaticTest`.
Inside the `Main` class, we can see that we are calling the non-static method using the object of the class (`st.multiply(2, 2)`). However, we are calling the static method by using the class name (`StaticTest.add(2, 3)`).

Static Variables

In Java, when we create objects of a class, then every object will have its own copy of all the variables of the class. For example,

```
class Test {
    // regular variable
   int age;
}

class Main {
    // create instances of Test
    Test test1 = new Test();
    Test test2 = new Test();
```

```
}
```

Here, both the objects test1 and test2 will have separate copies of the variable age. And, they are different from each other.

However, if we declare a variable static, all objects of the class share the same static variable. It is because like static methods, static variables are also associated with the class. And, we don't need to create objects of the class to access the static variables. For example,

```
class Test {
    // static variable
    static int age;
}
class Main {
    // access the static variable
    Test.age = 20;
}
```

Here, we can see that we are accessing the static variable from the other class using the class name.

Example 2: Java static and non-static Variables

```
class Test {

   // static variable
   static int max = 10;
```

```
    // non-static variable
    int min = 5;
}

public class Main {
    public static void main(String[] args) {
        Test obj = new Test();

        // access the non-static variable
        System.out.println("min + 1 = " +
(obj.min + 1));

        // access the static variable
        System.out.println("max + 1 = " +
(Test.max + 1));
    }
}
```

Output:

```
min + 1 = 6
max + 1 = 11
```

In the above program, we have declared a non-static variable named `min` and a static variable named `max` inside the class `Test`.

Inside the `Main` class, we can see that we are calling the non-static variable using the object of the class (`obj.min + 1`). However, we are calling the static variable by using the class name (`Test.max + 1`).

Note: Static variables are rarely used in Java. Instead, the static constants are used. These static constants are defined by `static final` keyword and represented in uppercase. This is why some people prefer to use uppercase for static variables as well.

Access static Variables and Methods within the Class

We are accessing the static variable from another class. Hence, we have used the class name to access it. However, if we want to access the static member from inside the class, it can be accessed directly. For example,

```
public class Main {

    // static variable
    static int age;
    // static method
    static void display() {
        System.out.println("Static Method");
    }
    public static void main(String[] args) {

        // access the static variable
        age = 30;
        System.out.println("Age is " + age);
```

```
        // access the static method
        display();
    }
}
```

Output:

```
Age is 30
Static Method
```

Here, we are able to access the static variable and method directly without using the class name. It is because static variables and methods are by default public. And, since we are accessing from the same class, we don't have to specify the class name.

Static Blocks

In Java, static blocks are used to initialize the static variables. For example,

```
class Test {
    // static variable
    static int age;

    // static block
    static {
```

```
        age = 23;
    }
}
```

Here we can see that we have used a static block with the syntax:

```
static {
    // variable initialization
}
```

The static block is executed only once when the class is loaded in memory. The class is loaded if either the object of the class is requested in code or the static members are requested in code.

A class can have multiple static blocks and each static block is executed in the same sequence in which they have been written in a program.

Example 3: Use of static block in java

```
class Main {

   // static variables
   static int a = 23;
   static int b;
   static int max;

   // static blocks
   static {
```

```
        System.out.println("First Static
block.");
        b = a * 4;
    }
    static {
        System.out.println("Second Static
block.");
        max = 30;
    }

    // static method
    static void display() {

        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("max = " + max);
    }

    public static void main(String args[]) {
        // calling the static method
        display();
    }
}
```

Output:

```
First Static block.
Second Static block.
a = 23
b = 92
```

```
max = 30
```

In the above program. as soon as the Main class is loaded,

- The value of `a` is set to `23`.
- The first static block is executed. Hence, the string `First Static block` is printed and the value of `b` is set to `a * 4`.
- The second static block is executed. Hence, the string `Second Static block` is printed and the value of `max` is set to `30`.
- And finally, the print statements inside the method `display()` are executed

Java Inheritance

In this tutorial, we will learn about Java inheritance and its types with the help of example.

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
}
```

In the above example, 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();

  }
}
```

Output

```
My name is Rohu
```

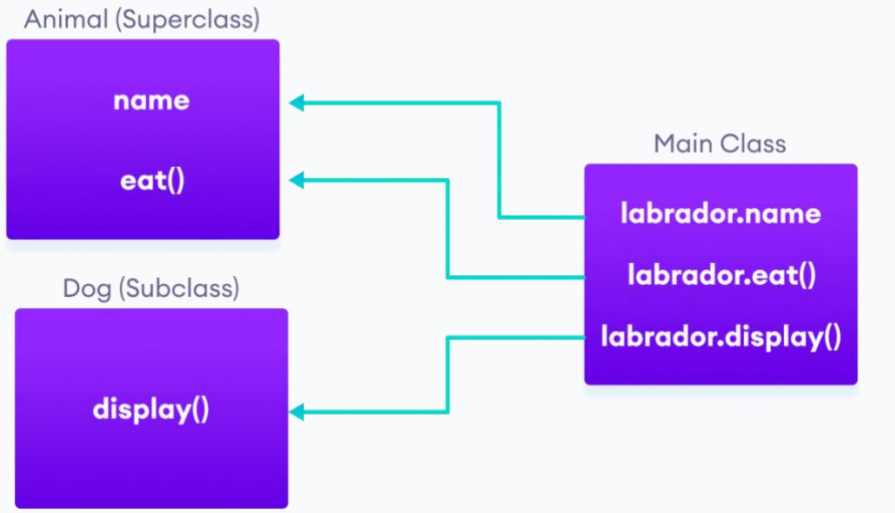
```
I can eat
```

In the above example, we have derived a subclass `Dog` from superclass `Animal`. Notice the statements,

```
labrador.name = "Rohu";

labrador.eat();
```

Here, `labrador` is an object of `Dog`.
However, `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`.



Java Inheritance Implementation

is-a relationship

In Java, inheritance is an **is-a**relationship. That is, we use inheritance only if there exists an is-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.
However, 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();
  }
```

```
}
```

Output

```
I eat dog food
I can bark
```

In the above example, 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. To learn more, visit Java 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();
  }
}
```

Output

```
I can eat
I eat dog food
I can bark
```

In the above example, 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. To learn more, visit Java super keyword.

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();
  }
}
```

Output

```
I am an animal.
My name is Rocky
```

In the above example, we have 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?

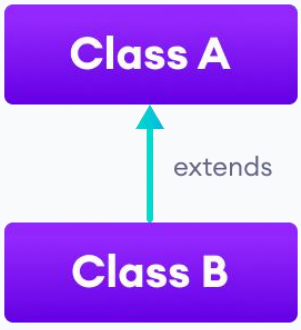
- The most important use of inheritance in Java is code reusability. The code that is present in the parent class can be directly used by the child class.
- Method overriding is also known as runtime polymorphism. Hence, we can achieve Polymorphism in Java with the help of inheritance.

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,

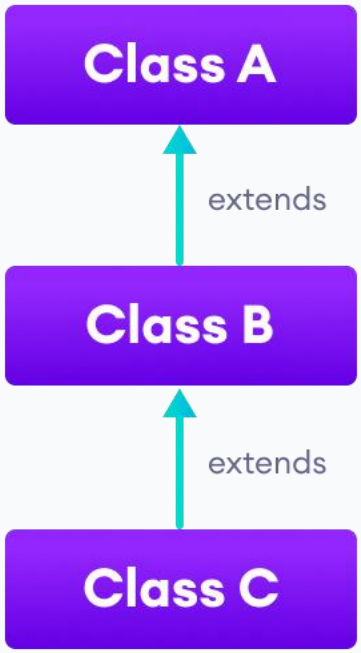


Java Single Inheritance

---

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,

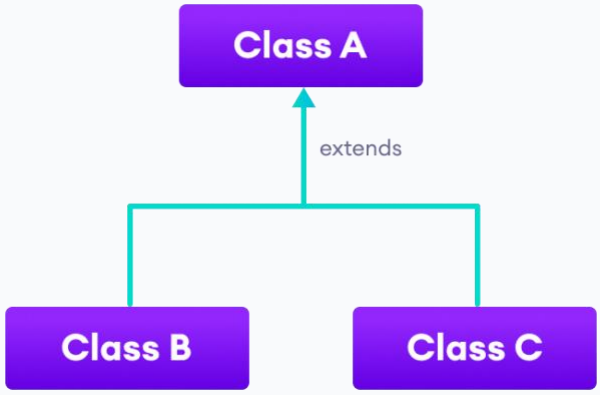


Java Multilevel Inheritance

---

3. Hierarchical Inheritance

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

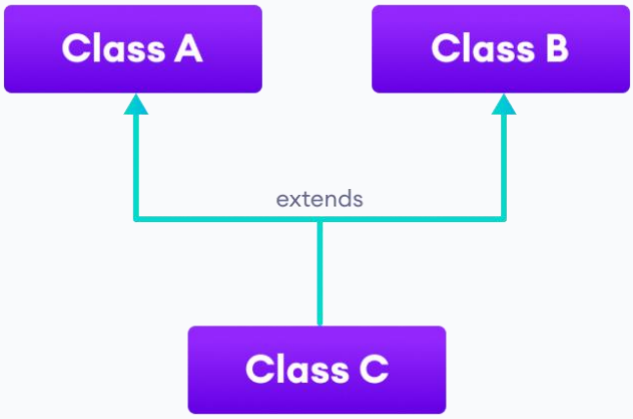


Java Hierarchical Inheritance

---

4. Multiple Inheritance

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



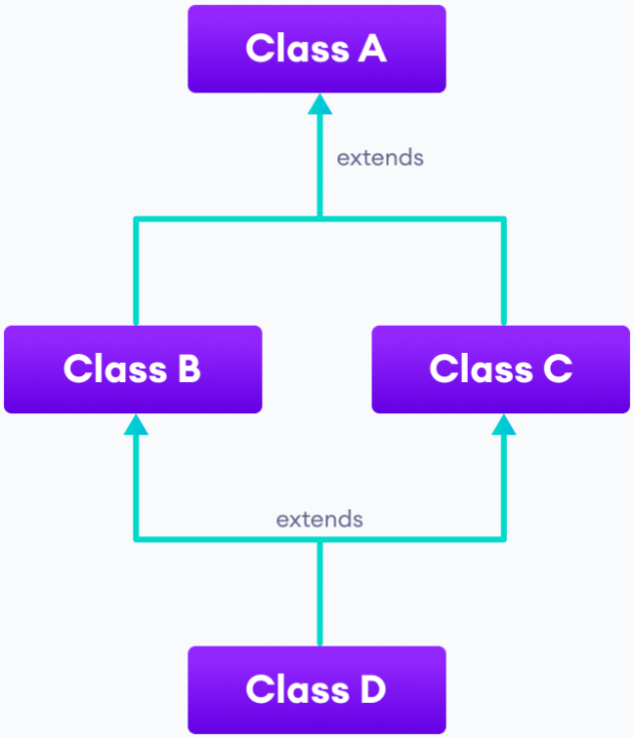
Java Multiple Inheritance

> **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

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

Java final keyword

In this tutorial, we will learn about Java final variables, methods and classes with examples.

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

In the above program, 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.

```
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

Before you learn about final methods and final classes, make sure you know about the Java Inheritance.
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();
  }
}
```

In the above example, we have created a final method named `display()` inside the `FinalDemo`class. Here, the `Main` class inherits the `FinalDemo` class.

We have 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 Arrays

In this tutorial, we will learn to work with arrays in Java. We will learn to declare, initialize, and access array elements with the help of examples.

An array is a collection of similar types of data.

For example, if we want to store the names of 100 people then we can create an array of the 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, here is how we can declare an array.

```
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 type `double`.

But, how many elements can array this hold?

Good question! To define the number of elements that an array can hold, we have to allocate memory for the array in Java. For example,

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

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

Here, the array can store **10**elements. We can also say that the **size or length** of the array is 10.
In Java, we can 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};
```

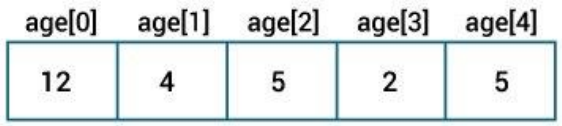
Here, we have created an array named age and initialized it with the values inside the curly brackets.

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;
..
```



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?

We can access the element of an array using the 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]);
  }
}
```

Output

```
Accessing Elements of Array:
First Element: 12
Second Element: 4
Third Element: 5
Fourth Element: 2
Fifth Element: 5
```

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]);
    }
  }
}
```

Output

```
Using for Loop:
12
4
5
```

In the above example, we are using the for Loop in Java 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);
   }
 }
}
```

Output

```
Using for-each Loop:
12
4
5
```

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);
 }
}
```

Output:

```
Sum = 36
Average = 3.6
```

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

```
int arrayLength = number.length;
```

Here, we are using the length attribute of the array to calculate the size of the array. We then calculate the average using:

```
average = ((double)sum / (double)arrayLength);
```

As you can see, we are converting the `int` value into `double`. This is called type casting in Java. To learn more about typecasting, visit Java Type Casting.

Multidimensional Arrays

Arrays we have mentioned till now are called one-dimensional arrays. However, we can declare multidimensional arrays in Java.

A multidimensional array is an array of arrays. That is, each 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. It is a 2-dimensional array

Java Multidimensional Arrays

In this tutorial, we will learn about the Java multidimensional array using 2-dimensional arrays and 3-dimensional arrays with the help of examples.

Before we learn about the multidimensional array, make sure you know about Java array.
A multidimensional array is an array of arrays. Each element of a multidimensional array is an array itself. For example,

```
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,

	Column 1	Column 2	Column 3	Column 4
Row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]

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];
```

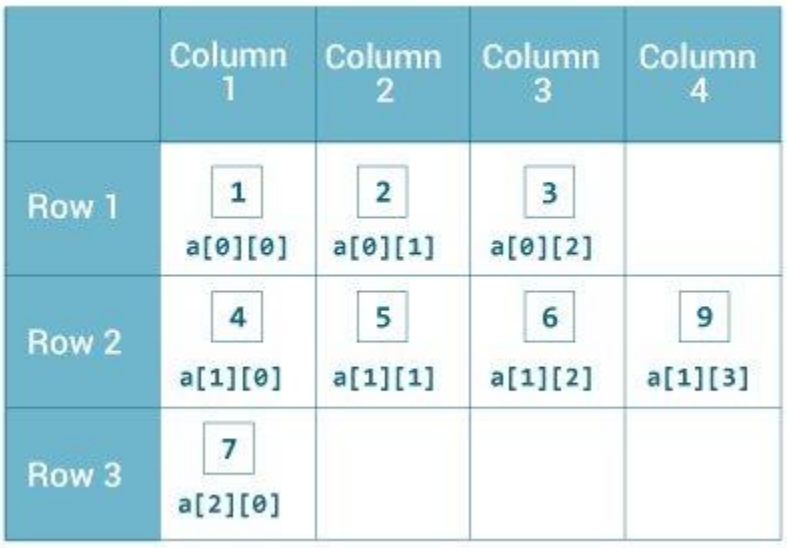
Here, `data` is a 3d array that can hold a maximum of 24 (3*4*2) elements of type `String`.

How to initialize a 2d array in Java?

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.



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);
    }
}
```

Output:

```
Length of row 1: 3
Length of row 2: 4
Length of row 3: 1
```

In the above example, we are creating a multidimensional array named `a`. Since each component of a multidimensional array is also 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 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]);
            }
        }
    }
}
```

Output:

```
1
-2
3
-4
-5
6
9
7
```

We can also use the for...each loopto access elements of the multidimensional array. For example,

```
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);
            }
        }
    }
}
```

Output:

```
1

-2
```

```
3
-4
-5
6
9
7
```

In the above example, we are have 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?

Let's see how we can use 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);
                }
            }
        }
    }
}
```

Output:

```
1
-2
3
2
3
4
-4
-5
6
9
1
2
3
```

Java String

In this tutorial, we will learn about Java Strings, how to create them, and various methods of String with the help of examples.

In Java, a string is a sequence of characters. For example, `"hello"` is a string containing a sequence of characters `'h'`, `'e'`, `'l'`, `'l'`, and `'o'`.

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`.

> **Note**: 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.

Example: Create a String in Java

```
class Main {
```

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

    // create strings
    String first = "Java";
    String second = "Python";
    String third = "JavaScript";

    // print strings
    System.out.println(first);   // print Java
    System.out.println(second);  // print
Python
    System.out.println(third);   // print
JavaScript
  }
}
```

In the above example, we have created three strings named `first`, `second`, and `third`. Here, we are directly creating strings like primitive types.

However, there is another way of creating Java strings (using the `new`keyword). We will learn about that later in this tutorial.

Java String Operations

Java String provides various methods to perform different operations on strings. We will look into some of the commonly used string operations.

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);
  }
}
```

Output

```
String: Hello! World
Length: 12
```

In the above example, the `length()` method calculates the total number of characters in the string and returns it. To learn more, visit Java String length().

2. Join two 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);
  }
}
```

Output

```
First String: Java
Second String: Programming
```

```
Joined String: Java Programming
```

In the above example, we have created two strings named `first`and `second`. Notice the statement,

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

Here, we the `concat()` method
joins `first` and `second` 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 comparisons between two strings using the `equals()` method. For example,

```
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);
  }
}
```

Output

```
Strings first and second are equal: true
Strings first and third are equal: false
```

In the above example, we have created 3 strings named `first`, `second`, and `third`. Here, we are using the `equal()` method to check if one string is equal to another.
The `equals()` method checks the content of strings while comparing them. To learn more, visit Java String equals().

> **Note**: We can also compare two strings using the `==`operator in Java. However, this approach is different than the `equals()` method. To learn more, visit Java String == vs equals().

Methods of Java String

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

Methods	Description
substring()	returns the substring of the string
replace()	replaces the specified old character with the specified new character
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

Escape character in Java Strings

The 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 will treat `"This is the "` as the string. Hence, 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 the string `"Hello! "`.

Now suppose we 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. However, this is not `true`.

Let's see what has happened here,

1. JVM takes the first string `"Hello! "`
2. creates a new string by adding `"World"` to the first string
3. assign the new string `"Hello! World"` to the `example` variable
4. 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. For example,

```
// create a string using the new keyword
```

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

In the above example, we have created a string `name` using the `new` keyword.

Here, when we create a string object, the `String()` constructor is invoked. To learn more about constructor, visit Java Constructor.

> **Note**: The `String` class provides various other constructors to create strings. To learn more, visit Java String (official Java documentation).

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
  }
}
```

Create String using literals vs new keyword

Now that we know how strings are created using string literals and the `new` keyword, let's see what is the major difference between them.

In Java, the JVM maintains a **string pool** to store all of its strings inside the 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, the new string is created all the time

Java Abstract Class and Abstract Methods

In this tutorial, we will learn about Java abstract classes and methods with the help of examples. We will also learn about abstraction in Java.

Java Abstract Class

The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the `abstract` keyword to declare an abstract class. For example,

```
// create an abstract class
abstract class Language {
  // fields and methods
}
...

// try to create an object Language
// throws an error
Language obj = new Language();
```

An abstract class can have both the regular methods and abstract methods. For example,

```
abstract class Language {

  // abstract method
  abstract void method1();
```

```
  // regular method
  void method2() {
    System.out.println("This is regular
method");
  }
}
```

To know about the non-abstract methods, visit Java methods. Here, we will learn about abstract methods.

Java Abstract Method

A method that doesn't have its body is known as an abstract method. We use the same `abstract` keyword to create abstract methods. For example,

```
abstract void display();
```

Here, `display()` is an abstract method. The body of `display()` is replaced by `;`.
If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. For example,

```
// error
// class should be abstract
class Language {
```

```
  // abstract method
  abstract void method1();
}
```

Example: Java Abstract Class and Method

Though abstract classes cannot be instantiated, we can create subclasses from it. We can then access members of the abstract class using the object of the subclass. For example,

```
abstract class Language {

  // method of abstract class
  public void display() {
    System.out.println("This is Java
Programming");
  }
}

class Main extends Language {

  public static void main(String[] args) {

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

```
    // access method of abstract class
    // using object of Main class
    obj.display();
  }
}
```

Output

```
This is Java programming
```

In the above example, we have created an abstract class named `Language`. The class contains a regular method `display()`.

We have created the Main class that inherits the abstract class. Notice the statement,

```
obj.display();
```

Here, `obj` is the object of the child class `Main`. We are calling the method of the abstract class using the object `obj`.

Implementing Abstract Methods

If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method.

For example,

```java
abstract class Animal {
  abstract void makeSound();

  public void eat() {
    System.out.println("I can eat.");
  }
}

class Dog extends Animal {

  // provide implementation of abstract method
  public void makeSound() {
    System.out.println("Bark bark");
  }
}

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

    // create an object of Dog class
    Dog d1 = new Dog();

    d1.makeSound();
    d1.eat();
  }
}
```

Output

```
Bark bark
```

```
I can eat.
```

In the above example, we have created an abstract class `Animal`. The class contains an abstract method `makeSound()` and a non-abstract method `eat()`. We have inherited a subclass `Dog`from the superclass `Animal`. Here, the subclass `Dog` provides the implementation for the abstract method `makeSound()`. We then used the object `d1` of the `Dog` class to call methods `makeSound()` and `eat()`.

> **Note**: If the `Dog` class doesn't provide the implementation of the abstract method `makeSound()`, `Dog` should also be declared as abstract. This is because the subclass `Dog`inherits `makeSound()` from `Animal`.

Java Interface

In this tutorial, we will learn about Java interfaces. We will learn how to implement interfaces and when to use them in detail with the help of examples.

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

We use the `interface` keyword to create an interface in Java. For example,

```
interface Language {
  public void getType();
```

```
  public void getVersion();
}
```

Here,

- `Language` is an interface.
- It includes abstract methods: `getType()` and `getVersion()`.

Implementing an Interface

Like abstract classes, we cannot create objects of interfaces.

To use an interface, other classes must implement it. We use the `implements` keyword to implement an interface.

Example 1: Java Interface

```
interface Polygon {
  void getArea(int length, int breadth);
}

// implement the Polygon interface
class Rectangle implements Polygon {

  // implementation of abstract method
  public void getArea(int length, int breadth)
{
```

```
    System.out.println("The area of the
rectangle is " + (length * breadth));
  }
}

class Main {
  public static void main(String[] args) {
    Rectangle r1 = new Rectangle();
    r1.getArea(5, 6);
  }
}
```

Output

```
The area of the rectangle is 30
```

In the above example, we have created an interface named `Polygon`. The interface contains an abstract method `getArea()`.
Here, the `Rectangle` class implements `Polygon`. And, provides the implementation of the `getArea()` method.

Example 2: Java Interface

```
// create an interface
interface Language {
  void getName(String name);
}
```

```
// class implements interface
class ProgrammingLanguage implements Language {

  // implementation of abstract method
  public void getName(String name) {
    System.out.println("Programming Language: "
+ name);
  }
}

class Main {
  public static void main(String[] args) {
    ProgrammingLanguage language = new
ProgrammingLanguage();
    language.getName("Java");
  }
}
```

Output

```
Programming Language: Java
```

In the above example, we have created an interface named `Language`. The interface includes an abstract method `getName()`.
Here, the `ProgrammingLanguage` class implements the interface and provides the implementation for the method.

Implementing Multiple Interfaces

In Java, a class can also implement multiple interfaces. For example,

```
interface A {
  // members of A
}

interface B {
  // members of B
}

class C implements A, B {
  // abstract members of A
  // abstract members of B
}
```

Extending an Interface

Similar to classes, interfaces can extend other interfaces. The `extends` keyword is used for extending interfaces. For example,

```
interface Line {
  // members of Line interface
}
```

```
// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

Here, the `Polygon` interface extends the `Line` interface. Now, if any class implements `Polygon`, it should provide implementations for all the abstract methods of both `Line` and `Polygon`.

Extending Multiple Interfaces

An interface can extend multiple interfaces. For example,

```
interface A {
   ...
}
interface B {
   ...
}

interface C extends A, B {
   ...
}
```

Advantages of Interface in Java

Now that we know what interfaces are, let's learn about why interfaces are used in Java.

- Similar to abstract classes, interfaces help us to achieve **abstraction in Java**.

  Here, we know `getArea()` calculates the area of polygons but the way area is calculated is different for different polygons. Hence, the implementation of `getArea()` is independent of one another.
- Interfaces **provide specifications** that a class (which implements it) must follow.

  In our previous example, we have used `getArea()` as a specification inside the interface `Polygon`. This is like setting a rule that we should be able to get the area of every polygon.

  Now any class that implements the `Polygon` interface must provide an implementation for the `getArea()` method.
- Interfaces are also used to achieve multiple inheritance in Java. For example,

```
interface Line {
…
}

interface Polygon {
…
}

class Rectangle implements Line, Polygon {
…
}
```

Here, the class `Rectangle` is implementing two different interfaces. This is how we achieve multiple inheritance in Java.

Note: All the methods inside an interface are implicitly `public` and all fields are implicitly `public static final`. For example,

```
interface Language {

  // by default public static final
  String type = "programming language";

  // by default public
  void getName();
```

```
}
```

---

default methods in Java Interfaces

With the release of Java 8, we can now add methods with implementation inside an interface. These methods are called default methods.

To declare default methods inside interfaces, we use the `default`keyword. For example,

```
public default void getSides() {
   // body of getSides()
}
```

Why default methods?

Let's take a scenario to understand why default methods are introduced in Java.

Suppose, we need to add a new method in an interface.

We can add the method in our interface easily without implementation. However, that's not the end of the story. All our classes that implement that interface must provide an implementation for the method.

If a large number of classes were implementing this interface, we need to track all these classes and make changes to them. This is not only tedious but error-prone as well.

To resolve this, Java introduced default methods. Default methods are inherited like ordinary methods.

Let's take an example to have a better understanding of default methods.

Example: Default Method in Java Interface

```
interface Polygon {
  void getArea();

  // default method
  default void getSides() {
    System.out.println("I can get sides of a
polygon.");
  }
}

// implements the interface
class Rectangle implements Polygon {
  public void getArea() {
    int length = 6;
    int breadth = 5;
```

```
    int area = length * breadth;
    System.out.println("The area of the
rectangle is " + area);
  }

  // overrides the getSides()
  public void getSides() {
    System.out.println("I have 4 sides.");
  }
}

// implements the interface
class Square implements Polygon {
  public void getArea() {
    int length = 5;
    int area = length * length;
    System.out.println("The area of the square
is " + area);
  }
}

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

    // create an object of Rectangle
    Rectangle r1 = new Rectangle();
    r1.getArea();
    r1.getSides();

    // create an object of Square
```

```
    Square s1 = new Square();
    s1.getArea();
    s1.getSides();
  }
}
```

Output

```
The area of the rectangle is 30
I have 4 sides.
The area of the square is 25
I can get sides of a polygon.
```

In the above example, we have created an interface named `Polygon`. It has a default method `getSides()` and an abstract method `getArea()`.

Here, we have created two classes `Rectangle` and `Square` that implement `Polygon`. The `Rectangle` class provides the implementation of the `getArea()` method and overrides the `getSides()` method. However, the `Square` class only provides the implementation of the `getArea()` method.

Now, while calling the `getSides()` method using the `Rectangle` object, the overridden method is called. However, in the case of the `Square` object, the default method is called.

---

private and static Methods in Interface

The Java 8 also added another feature to include static methods inside an interface.

Similar to a class, we can access static methods of an interface using its references. For example,

```
// create an interface
interface Polygon {
  staticMethod(){..}
}

// access static method
Polygon.staticMethod();
```

> **Note**: With the release of Java 9, private methods are also supported in interfaces.
> We cannot create objects of an interface. Hence, private methods are used as helper methods that provide support to other methods in interfaces.

Practical Example of Interface

Let's see a more practical example of Java Interface.

```
// To use the sqrt function
```

```
import java.lang.Math;

interface  Polygon {
   void getArea();

 // calculate the perimeter of a Polygon
   default void getPerimeter(int... sides) {
      int perimeter = 0;
      for (int side: sides) {
         perimeter += side;
      }

   System.out.println("Perimeter: " +
perimeter);
   }
}

class Triangle implements Polygon {
   private int a, b, c;
   private double s, area;

// initializing sides of a triangle
   Triangle(int a, int b, int c) {
      this.a = a;
      this.b = b;
      this.c = c;
      s = 0;
   }

// calculate the area of a triangle
```

```
    public void getArea() {
        s = (double) (a + b + c)/2;
        area = Math.sqrt(s*(s-a)*(s-b)*(s-c));
        System.out.println("Area: " + area);
    }
}

class Main {
    public static void main(String[] args) {
        Triangle t1 = new Triangle(2, 3, 4);

// calls the method of the Triangle class
        t1.getArea();

// calls the method of Polygon
        t1.getPerimeter(2, 3, 4);
    }
}
```

Output

```
Area: 2.9047375096555625
Perimeter: 9
```

In the above program, we have created an interface named `Polygon`. It includes a default method `getPerimeter()` and an abstract method `getArea()`.

We can calculate the perimeter of all polygons in the same manner so we implemented the body of `getPerimeter()` in `Polygon`.

Now, all polygons that implement `Polygon` can use `getPerimeter()` to calculate perimeter.
However, the rule for calculating the area is different for different polygons. Hence, `getArea()` is included without implementation.
Any class that implements `Polygon`must provide an implementation of `getArea()`.