

Java Class and Objects

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

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

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

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

Java Class

A class is a blueprint for the object. Before we create an object, we first need to define the class.

We can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

Create a class in Java

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

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

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

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

For our *bicycle* object, we can create the class as

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

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

Here, *Bicycle* is a prototype. Now, we can create any number of bicycles using the prototype. And, all the bicycles will share the fields and methods of the prototype.

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

Java Objects

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

Creating an Object in Java

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

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

We have used the *new* keyword along with the constructor of the class to create an object.

Constructors are similar to methods and have the same name as the class. For example, *Bicycle()* is the constructor of the *Bicycle* class.

Here, *sportsBicycle* and *touringBicycle* are the names of objects. We can use them to access fields and methods of the class.

As you can see, we have created two objects of the class. We can create multiple objects of a single class in Java.

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

Access Members of a Class

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

```
class Bicycle {  
  
    // field of class  
    int gear = 5;  
  
    // method of class  
    void braking() {  
        ...  
    }  
}  
  
// create object  
Bicycle sportsBicycle = new Bicycle();  
  
// access field and method  
sportsBicycle.gear;  
sportsBicycle.braking();
```

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

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 quite a few times.

Now that we understand what is class and object. Let's see a fully working example.

Example: Java Class and Objects

```
class Lamp {

    // stores the value for light
    // true if light is on
    // false if light is off
    boolean isOn;

    // method to turn on the light
    void turnOn() {
        isOn = true;
        System.out.println("Light on? " + isOn);
    }

    // method to turnoff the light
    void turnOff() {
        isOn = false;
        System.out.println("Light on? " + isOn);
    }
}

class 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();
    }
}
```

Output:

```
Light on? true
Light on? false
```

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

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

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

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

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

Example: Create objects inside the same class

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

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

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

Output

```
Light on? true
```

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

Java Methods

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

Suppose you need to create a program to create a circle and color it. You can create two methods to solve this problem:

- a method to draw the circle
- a method to color the circle

Dividing a complex problem into smaller chunks makes your program easy to understand and reusable.

In Java, there are two types of methods:

- **User-defined Methods:** We can create our own method based on our requirements.
- **Standard Library Methods:** These are built-in methods in Java that are available to use.

Declaring a Java Method

The syntax to declare a method is:

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

Here,

- **returnType** - It specifies what type of value a method returns. For example, if a method has an int return type, then it returns an integer value.

If the 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 addNumbers(). And, the return type is int. We will learn more about return types later in this tutorial.

This is the simple syntax of declaring a method. However, the 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.

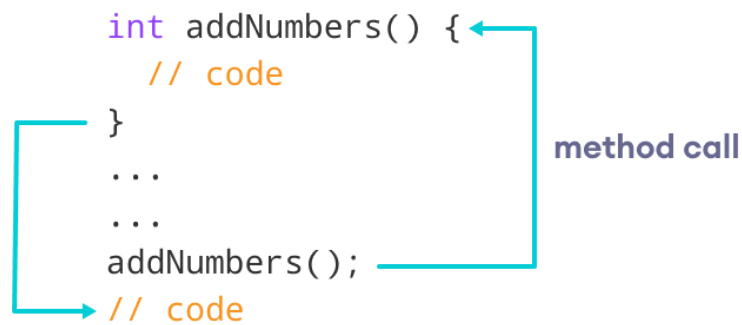
- **parameter1/parameter2** - These are values passed to a method. We can pass any number of arguments to a method.

Calling a Method in Java

In the above example, we have 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();
```



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);  
    }  
}
```

Output

Sum is: 40

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 have 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 the 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);  
    }  
}
```

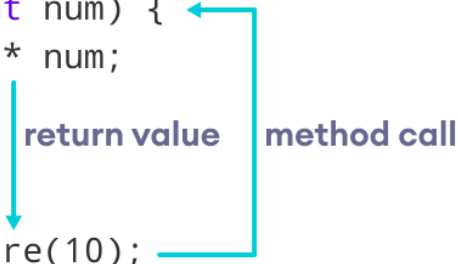
Output:

```
Squared value of 10 is: 100
```

In the above program, we have 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.

```
int square(int num) {  
    return num * num;  
}  
...  
...  
result = square(10);  
// code
```



The diagram illustrates the flow of a return value. A blue arrow labeled "return value" points from the `return` statement in the `square` method to the `square(10)` call in the code below. Another blue arrow labeled "method call" points from the `square(10)` call back to the `square` method definition.

Representation of the Java method returning a value

Note: If the 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: " + a);  
}
```

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);
    }
}
```

Output

```
Method without parameter
Method with a single parameter: 24
```

Here, the parameter of the method is `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 the method call is called the actual argument.

The parameter *num* accepted by the method definition is known as a 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. These 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.PrintStream`. 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));  
    }  
}
```

Output:

Square root of 4 is: 2.0

What are the advantages of using methods?

1. The main advantage is **code reusability**. We can write a method once, and use it multiple times. We do not have to rewrite the entire code each time. Think of it as, "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);
    }
}

```

Output:

```

Square of 1 is: 1
Square of 2 is: 4
Square of 3 is: 9
Square of 4 is: 16
Square of 5 is: 25

```

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

Hence, 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) { ... }

```

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 arguments

```
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. 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.

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.

Why do constructors not return values?

What actually happens with the constructor is that the runtime uses type data generated by the compiler to determine how much space is needed to store an object instance in memory, be it on the stack or on the heap.

This space includes all members variables and the vtbl. After this space is allocated, the constructor is called as an internal part of the instantiation and initialization process to initialize the contents of the fields.

Then, when the constructor exits, the runtime returns the newly-created instance. So the reason the constructor doesn't return a value is because it's not called directly by your code, it's called by the memory allocation and object initialization code in the runtime.

Its return value (if it actually has one when compiled down to machine code) is opaque to the user - therefore, you can't specify it.

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.

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
```

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);
}
}

```

Output:

```

a = 0
b = false

```

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 Language: " + 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.

It is also possible to call one constructor from another constructor.

Note: We have used `this` keyword to specify the variable of the class.

Java String

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.

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 use the `concat()` method to join *first* and *second* and assign it to the *joinedString* variable.

We can also join two strings using the `+` operator in Java.

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.

Note: We can also compare two strings using the `==` operator in Java. However, this approach is different than the `equals()` method.

Methods of Java String

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

Methods	Description
<code>substring()</code>	returns the substring of the string
<code>replace()</code>	replaces the specified old character with the specified new character
<code>charAt()</code>	returns the character present in the specified location
<code>getBytes()</code>	converts the string to an array of bytes
<code>indexOf()</code>	returns the position of the specified character in the string
<code>compareTo()</code>	compares two strings in the dictionary order
<code>trim()</code>	removes any leading and trailing whitespaces
<code>format()</code>	returns a formatted string
<code>split()</code>	breaks the string into an array of strings
<code>toLowerCase()</code>	converts the string to lowercase
<code>toUpperCase()</code>	converts the string to uppercase
<code>valueOf()</code>	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 string 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.

Note: 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
    }
}
```

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 Access Modifiers

What are 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 can not be accessed by other classes.

Note the keyword `public` and `private`. These are access modifiers in Java. 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

Default declarations are visible only within the package (package private)

Private declarations are visible within the class only

Protected declarations are visible within the package or all subclasses

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. However, if we try to use the *Logger* class in another class outside of *defaultPackage*, we 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 = "Programiz";
    }
}
```

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

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

The error is generated because we are trying to access the private variable and the private method 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 method. 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("Programiz");
        System.out.println(d.getName());
    }
}

```

Output:

The name is Programiz

In the above example, we have a private variable named *name*. In order to access the variable from the outer class, we have used methods: `getName()` and `setName()`. These methods are called getter and setter in Java.

Here, we have used the setter method (`setName()`) to assign value to the variable and the getter method (`getName()`) to access the variable.

We have used this keyword inside the `setName()` to refer to the variable of the class.

Note: We cannot 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();
    }
}

```

```
    }  
}
```

Output:

```
I am an animal
```

In the above example, 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();
    }
}

```

Output:

```

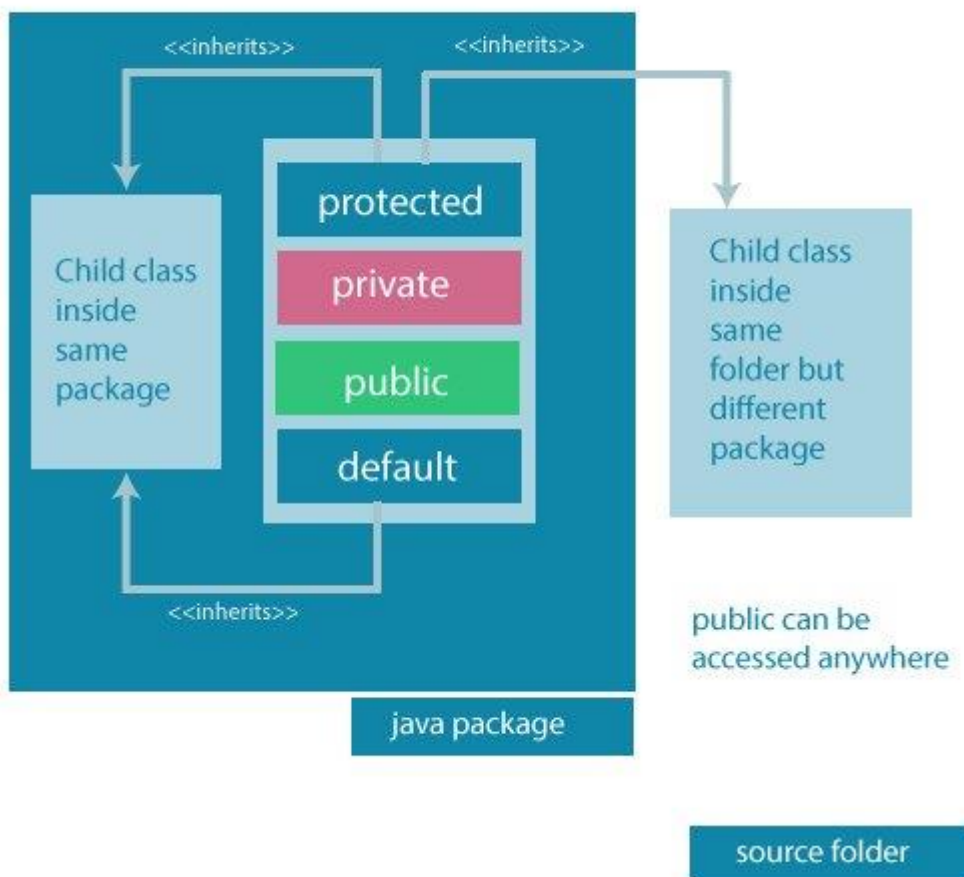
I am an animal.
I have 4 legs.

```

Here,

- 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 can 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

this Keyword

In Java, this keyword is used to refer to the current object inside a method or a constructor. For example,

```
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);
    }
}
```

Output:

```
this reference = Main@23fc625e
object reference = Main@23fc625e
```

In the above example, 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 this is nothing but the reference to the current object.

Use of 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). However, 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);  
    }  
}
```

Output:

```
mc.age = 0
```

In the above example, 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);
    }
}

```

Output:

```
obj.age = 8
```

Now, we are getting the expected output. It is because when the constructor is called, this inside the constructor is replaced by the object obj that has called the constructor. Hence the age variable is assigned value 8.

Also, 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;
    }
}

```

this with Getters and Setters

Another common use of this keyword is in *setters and getters methods* of a class. For example:

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

Output:

```
obj.name: Toshiba
```

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

While working with constructor overloading, we might 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()`. Let's take an example,

```

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);
    }
}

```

Output:

2 + 3i


```
3 + 3i  
0 + 0i
```

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. However, we should be always careful while using `this()`.

This is because calling 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);
    }
}

```

Output:

```

Before passing this to addTwo() method:
x = 1, y = -2
After passing this to addTwo() method:
x = 3, y = 0

```

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

```
add(this);
```

Here, we are calling 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.

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

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 Recursion

In Java, a method that calls itself is known as a recursive method. And, this process is known as recursion.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

How Recursion works?

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

static void recurse() {
    ... ..
    recurse()
    ... ..
}
```

Normal Method Call

Recursive Call

Working of Java Recursion

In the above example, 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);  
    }  
}
```

Output:

```
4 factorial = 24
```

In the above example, 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,

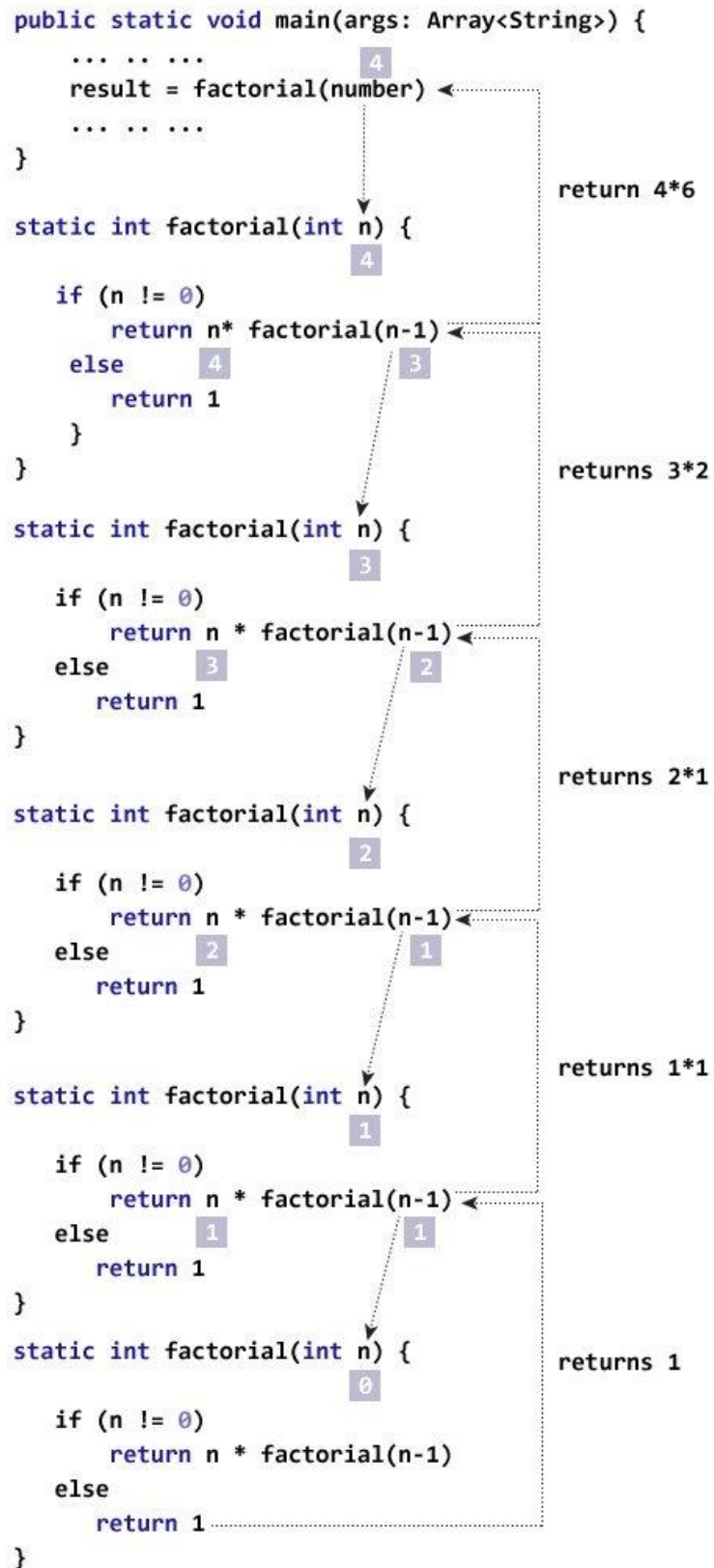
```
return n * factorial(n-1);
```

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.

Working of Factorial Program

The image below will give you a better idea of how the factorial program is executed using recursion.



Factorial Program using Recursion

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.

Any algorithm implemented using recursion can also be implemented using iteration.

Why *not* to use recursion

1. It is usually slower due to the overhead of maintaining the stack.
2. It usually uses more memory for the stack.

Why *to* use recursion

1. Recursion adds clarity and (sometimes) reduces the time needed to write and debug code (but doesn't necessarily reduce space requirements or speed of execution).
2. Reduces time complexity.
3. Performs better in solving problems based on tree structures.

For example, the Tower of Hanoi problem is more easily solved using recursion as opposed to iteration.

Java instanceof Operator

The 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. Otherwise, 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);  
    }  
}
```

Output:

```
name is an instance of String: true  
obj is an instance of Main: true
```

In the above example, 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.

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
    }
}
```

In the above example, 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 instanceof in Interface

The instanceof operator is also used to check whether an object of a class is also an instance of the interface implemented by the class. For example,

```
// Java program to check if an object of a class is also
```

```
// an instance of the interface implemented by the class

interface Animal {
}

class Dog implements Animal {
}

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

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

        // checks if the object of Dog
        // is also an instance of Animal
        System.out.println(d1 instanceof Animal); // returns true
    }
}
```

In the above example, the *Dog* class implements the *Animal* interface. Inside the print statement, notice the expression,

```
d1 instanceof Animal
```

Here, *d1* is an instance of *Dog* class. The instanceof operator checks if *d1* is also an instance of the interface *Animal*.

Note: In Java, all the classes are inherited from the Object class. So, instances of all the classes are also an instance of the Object class.

In the previous example, if we check,

```
d1 instanceof Object
```

The result will be true.