
Java this Keyword

In this article, we will learn about this keyword in Java, how and where to use them with the help of examples.

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 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 classfinal class FinalClass {
  public void display() {
    System.out.println("This is a final method.");
  }
}
// try to extend the final classclass 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 FinalClassclass Main extends FinalClass {
                  ^
```

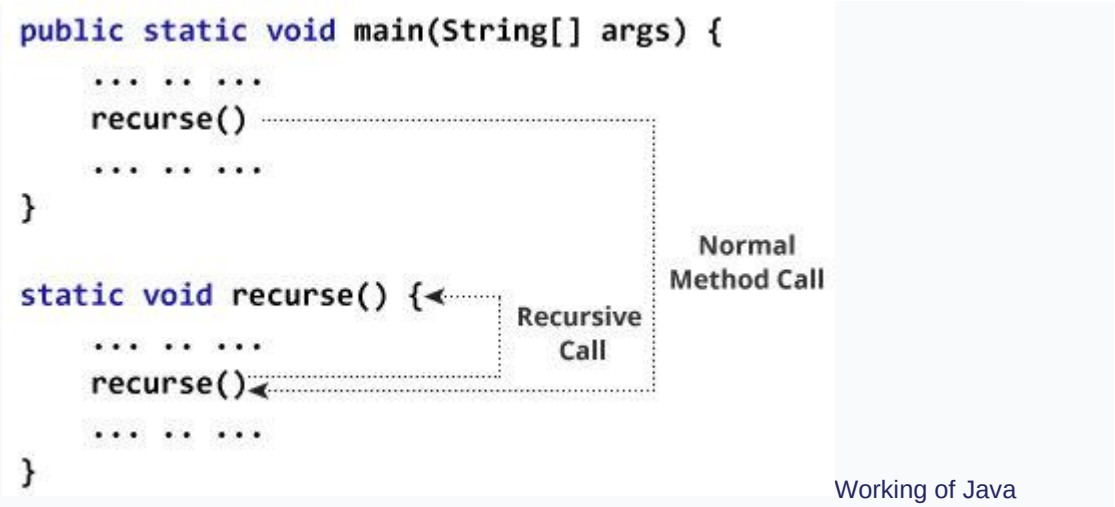
Java Recursion

In this tutorial, you will learn about Java recursive function, its advantages and disadvantages.

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?



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,

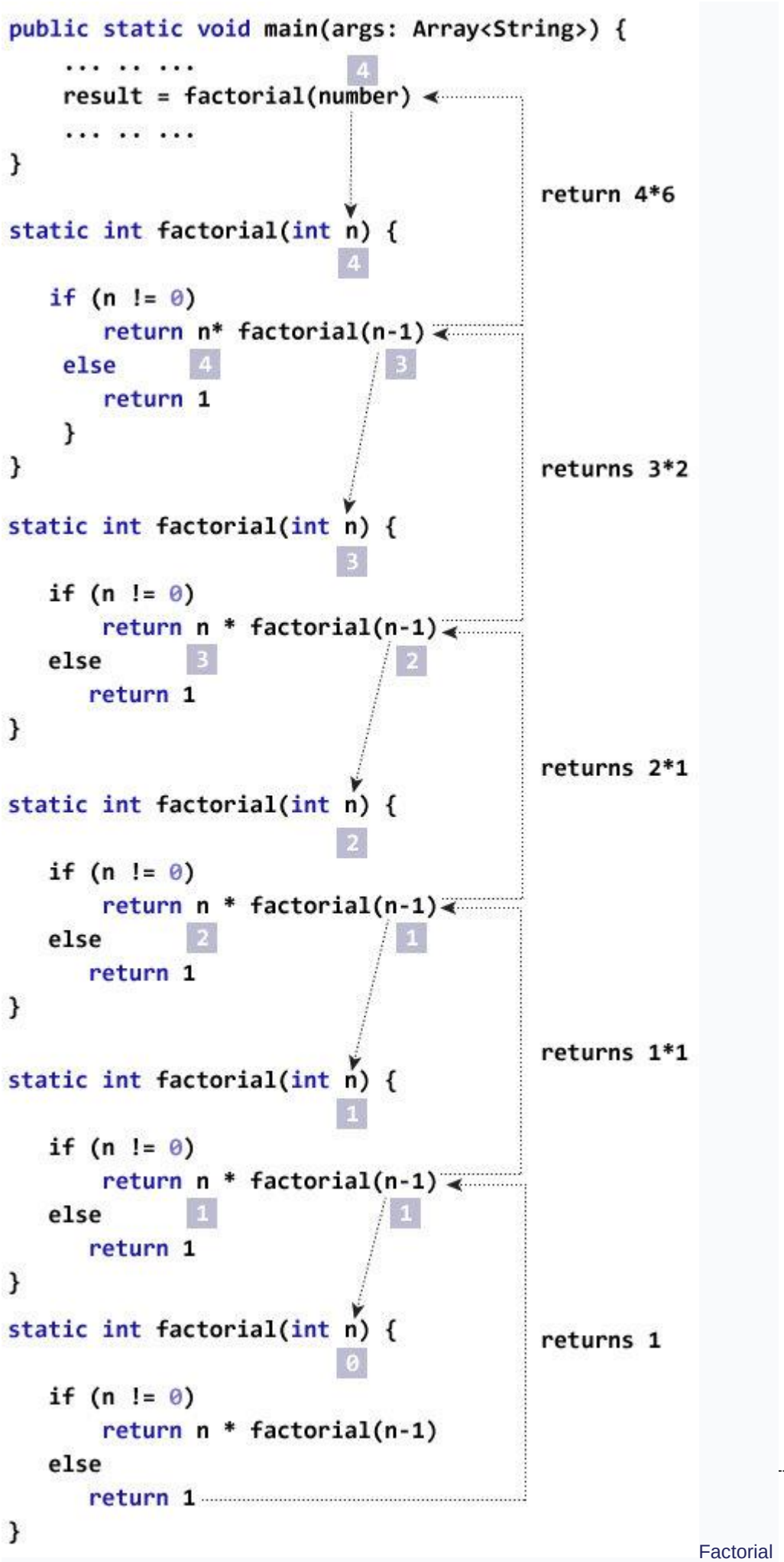
```
return n * factorial(n-1);
```

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.


public static void main(args: Array<String>) {
... .
result = factorial(number)
... .
}
4
return 4*6
static int factorial(int n) {
4
if (n != 0)
return n* factorial(n-1)
else
return 1
}
4
3
returns 3*2
static int factorial(int n) {
3
if (n != 0)
return n * factorial(n-1)
else
return 1
}
3
2
returns 2*1
static int factorial(int n) {
2
if (n != 0)
return n * factorial(n-1)
else
return 1
}
2
1
returns 1*1
static int factorial(int n) {
1
if (n != 0)
return n * factorial(n-1)
else
return 1
}
1
1
static int factorial(int n) {
0
if (n != 0)
return n * factorial(n-1)
else
return 1
}
returns 1

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.

Java instanceof Operator

In this tutorial, you will learn about Java instanceof operator in detail with the help of examples.

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. To learn more, visit Java String.

Java instanceof during Inheritance

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

---

```
// Java Program to check if an object of the subclass// is also an instance of the superclass

// superclassclass Animal {

}

// subclassclass 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`.
