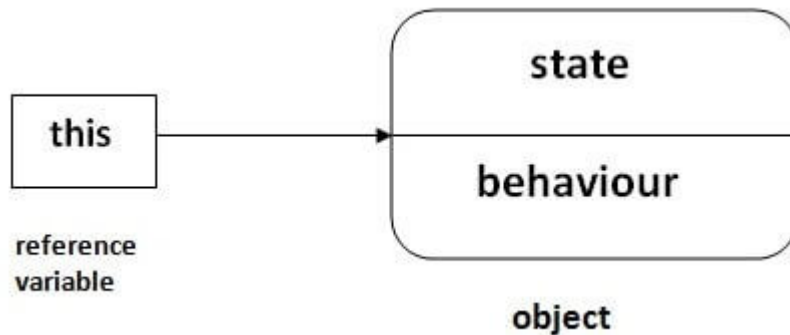


## this keyword in Java

There can be a lot of usage of **Java this keyword**. In Java, this is a **reference variable** that refers to the current object.



### Usage of Java this Keyword

There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.

01

**this** can be used to refer current class instance variable.

04

**this** can be passed as an argument in the method call.

02

**this** can be used to invoke current class method (implicity)

05

**this** can be passed as argument in the constructor call.

03

**this()** can be used to invoke current class Constructor.

06

**this** can be used to return the current class instance from the method

### 1) **this**: to refer current class instance variable

The **this** keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

we are using this keyword to distinguish local variable and instance variable.

**class** Student

```
{  
    int rollno;  
    String name;  
    float fee;  
    Student(int rollno,String name,float fee)  
    {  
        this.rollno=rollno;  
        this.name=name;  
        this.fee=fee;  
    }  
    void display()  
    {  
        System.out.println(rollno+" "+name+" "+fee);  
    }  
}
```

**class** TestThis2

```
{  
    public static void main(String args[])  
    {  
        Student s1=new Student(111,"ankit",5000f);
```

```
        Student s2=new Student(112,"sumit",6000f);  
        s1.display();  
        s2.display();  
    }  
}
```

**Output:**

```
111 ankit 5000.0  
112 sumit 6000.0
```

## Access Modifiers in Java

1. Private access modifier
2. Role of private constructor
3. Default access modifier
4. Protected access modifier
5. Public access modifier
6. Access Modifier with Method Overriding

There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it.

There are four types of Java access modifiers:

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

2. **Default:** The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

There are many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient, etc. Here, we are going to learn the access modifiers only.

### Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

| Access Modifier | within class | within package | outside package by subclass only | outside package |
|-----------------|--------------|----------------|----------------------------------|-----------------|
| Private         | Y            | N              | N                                | N               |
| Default         | Y            | Y              | N                                | N               |
| Protected       | Y            | Y              | Y                                | N               |
| Public          | Y            | Y              | Y                                | Y               |

#### 1) Private

The private access modifier is accessible only within the class.

#### Simple example of private access modifier

In this example, we have created two classes A and Simple. A class contains private data member and private method. We are accessing these private members from outside the class, so there is a compile-time error.

```
class A
{
private int data=40;
private void msg()
{
    System.out.println("Hello java");
}
}

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

### **Role of Private Constructor**

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

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

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

```

{
    A obj=new A();//Compile Time Error
}
}

```

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

## 2) Default

If you don't use any modifier, it is treated as **default** by default. The default modifier is accessible only within package. It cannot be accessed from outside the package. It provides more accessibility than private. But, it is more restrictive than protected, and public.

### Example of default access modifier

In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.

//save by A.java

**package** pack;

**class** A

```

{
    void msg()
    {
        System.out.println("Hello");
    }
}

```

//save by B.java

**package** mypack;

**import** pack.\*;

**class** B

```

{
    public static void main(String args[])
    {
        A obj = new A();//Compile Time Error
        obj.msg();//Compile Time Error
    }
}

```

```
}
```

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

### 3) Protected

The **protected access modifier** is accessible within package and outside the package but through inheritance only.

The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.

It provides more accessibility than the default modifier.

#### Example of protected access modifier

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

```
//save by A.java
```

```
package pack;
```

```
public class A
```

```
{
```

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

```
}
```

```
//save by B.java
```

```
package mypack;
```

```
import pack.*;
```

```
class B extends A
```

```
{
```

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

```
    B obj = new B();
```

```
    obj.msg();
```

```
}
```

```
}
```

```
Output:Hello
```

#### 4) Public

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

##### Example of public access modifier

```
//save by A.java
```

```
package pack;
```

```
public class A
```

```
{
```

```
public void msg(){System.out.println("Hello");}
```

```
}
```

```
//save by B.java
```

```
package mypack;
```

```
import pack.*;
```

```
class B
```

```
{
```

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

```
{
```

```
    A obj = new A();
```

```
    obj.msg();
```

```
}
```

```
}
```

```
Output:Hello
```

#### Java Access Modifiers with Method Overriding

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

```
class A
```

```
{
```

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



```
}
}
```

```
public class Simple extends A
```

```
{
```

```
void msg(){System.out.println("Hello java");} //C.T.Error
```

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

```
    Simple obj=new Simple();
```

```
    obj.msg();
```

```
}
```

```
}
```

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

### Difference between final, finally and finalize

The final, finally, and finalize are keywords in Java that are used in exception handling. Each of these keywords has a different functionality. The basic difference between final, finally and finalize is that the **final** is an access modifier, **finally** is the block in Exception Handling and **finalize** is the method of object class.

Along with this, there are many differences between final, finally and finalize. A list of differences between final, finally and finalize are given below:

| Sr. no. | Key        | final  | finally  | finalize   |
|---------|------------|--|--|--|
| 1.      | Definition | final is the keyword and access modifier which is used to apply restrictions on a class, method or variable. | finally is the block in Java Exception Handling to execute the important code whether the exception occurs or not. | finalize is the method in Java which is used to perform clean up processing just before object is garbage collected. |

|    |               |   |   |   |
|----|---------------|---|---|---|
| 2. | Applicable to | Final keyword is used with the classes, methods and variables.  | Finally block is always related to the try and catch block in exception handling.   | finalize() method is used with the objects.   |
| 3. | Functionality | (1) Once declared, final variable becomes constant and cannot be modified.<br>(2) final method cannot be overridden by sub class.<br>(3) final class cannot be inherited. | (1) finally block runs the important code even if exception occurs or not.<br>(2) finally block cleans up all the resources used in try block | finalize method performs the cleaning activities with respect to the object before its destruction. |
| 4. | Execution     | Final method is executed only when we call it.  | Finally block is executed as soon as the try-catch block is executed.<br>It's execution is not dependant on the exception.                    | finalize method is executed just before the object is destroyed.                                    |

### ***Java final Example***

Let's consider the following example where we declare final variable age. Once declared it cannot be modified.

#### **FinalExampleTest.java**

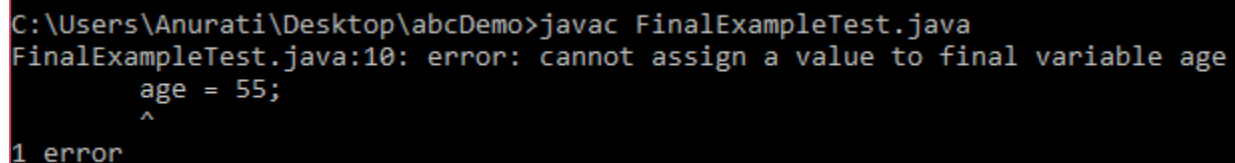
```
public class FinalExampleTest
{
    //declaring final variable
    final int age = 18;
    void display()
    {
        // reassigning value to age variable
```

```

// gives compile time error
age = 55;
}
public static void main(String[] args)
{
    FinalExampleTest obj = new FinalExampleTest();
    // gives compile time error
    obj.display();
}
}

```

### Output:



```

C:\Users\Anurati\Desktop\abcDemo>javac FinalExampleTest.java
FinalExampleTest.java:10: error: cannot assign a value to final variable age
    age = 55;
    ^
1 error

```

In the above example, we have declared a variable final. Similarly, we can declare the methods and classes final using the final keyword.

### *Java finally Example*

Let's see the below example where the Java code throws an exception and the catch block handles that exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

#### **FinallyExample.java**

```

public class FinallyExample
{
    public static void main(String args[])
    {
        try {
            System.out.println("Inside try block");
            // below code throws divide by zero exception
            int data=25/0;
            System.out.println(data);
        }
        // handles the Arithmetic Exception / Divide by zero exception
    }
}

```

```

catch (ArithmeticException e){
    System.out.println("Exception handled");
    System.out.println(e);
}
// executes regardless of exception occurred or not
finally
{
    System.out.println("finally block is always executed");
}
System.out.println("rest of the code...");
}
}

```

### Output:

```

C:\Users\Anurati\Desktop\abcDemo>java FinallyExample.java
Inside try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code...

```

### *Java finalize Example*

#### FinalizeExample.java

```

public class FinalizeExample
{
    public static void main(String[] args)
    {
        FinalizeExample obj = new FinalizeExample();
        // printing the hashCode
        System.out.println("HashCode is: " + obj.hashCode());
        obj = null;
        // calling the garbage collector using gc()
        System.gc();
        System.out.println("End of the garbage collection");
    }
    // defining the finalize method
    protected void finalize()

```

```
{  
    System.out.println("Called the finalize() method");  
}  
}
```

**Output:**

```
C:\Users\Anurati\Desktop\abcDemo>javac FinalizeExample.java  
Note: FinalizeExample.java uses or overrides a deprecated API.  
Note: Recompile with -Xlint:deprecation for details.  
  
C:\Users\Anurati\Desktop\abcDemo>java FinalizeExample  
Hashcode is: 746292446  
End of the garbage collection  
Called the finalize() method
```