Object class in Java

**What is object class in Java?** In Java, the Object class is the root or superclass of all classes. Directly or indirectly every predefined and user-defined Java class is a subclass of the Object class. Object class contains the most commonly required methods for every Java class.

This class name is chosen as Object, because as per coding standards, generally name will be chosen based on the operations it is performing. This class has methods to perform operations related to the object of a class, so its name is chosen as an Object.

All classes in Java are inherited from which class?

All classes in java are inherited from java.lang.Object class. Every class in Java is the child class of java.lang.Object class either directly or indirectly. Therefore, Object class methods are directly available for every Java class. Hence Object class is considered as the root of all Java classes.

class A {}

class B extends A {}

Here class A is a direct subclass of the Object class. Class B is a subclass of A, so indirectly B class is also the subclass of Object.

If our class doesn’t extend any class then only, our class is the direct child class of the Object class. For example, in the above case, class “A” is the direct subclass of the Object class. But if our class extends any other class then our class is an indirect child class of the Object class. For example in the above case, B is the subclass of A, and class A is the subclass of the Object class. Hence it is an indirect child class of the Object class. It is called a multi-level inheritance.

**class A { }**

**class A extends Object { }**

**class A extends java.lang.Object { }**

### **Reasons to Create Object Class as Super Class**

Object class is created as the superclass for all java classes due to below 2 reasons,

1. **To achieve reusability to subclasses**:- Every object will have some common behaviors. These common behaviors must be implemented in every class with the same signature by every class developer. So for informing these methods signature and to reduce the burden in implementing these methods, SUN developed a class called Object by implementing all these common behaviors with some methods common to all classes. All these methods have generic logic common for all subclasses. If this logic is not satisfying to subclass requirement then the subclass should override these methods.

2. **To achieve loose coupling and runtime polymorphism in user class**:- If we want to define a method receive and to return any type of object, and further to invoke and execute from this passed object class, we must have one common superclass to all classes.

bject Class Methods in Java

Every object will contain some common operations. To perform these operations separate methods are given.

| **Methods** | **Description** |
| --- | --- |
| [**Class<?> getClass()**](https://www.knowprogram.com/java/getclass-method-java/) | Returns the runtime class of this Object. |
| [**int hashCode()**](https://www.knowprogram.com/java/hashcode-method/) | Returns a hash code value for the object. |
| [**boolean equals(Object obj)**](https://www.knowprogram.com/java/equals-method-java/) | It is used to compare two objects. It returns true if this object is the same as the obj argument; false otherwise. |
| [**String toString()**](https://www.knowprogram.com/java/object-tostring-java/) | Returns a string representation of the object. |
| [**Object clone()**](https://www.knowprogram.com/java/clone-method-java/) | Creates and returns a copy of this object. |
| void**finalize()** | It is used to execute object clean-up code just before the object is being destroyed. |
| void**notify()**, & void**notifyAll()** | Notifying about object lock availability to waiting threads |
| void**wait()**, void**wait(long mills)**, & void **wait(long mills, int Nanos)** | Releasing object lock and sending thread to waiting-state |

These methods are discussed in detail in their own topic, all links are given. The finalize() method is related to the garbage collection features of Java. The notify(), notifyAll(), and wait() methods are related to the multi-threading feature of Java OOP.

## Object getClass() Method in Java

**Object getClass() Method in Java** | We can use getClass() method of java.lang.Object class to get the runtime class definition of a runtime class object. Prototype of getClass() method of [java.lang.Object class](https://www.knowprogram.com/java/object-class-java/) is:-

***public final native Class<?> getClass()***

Return value:- The Class object that represents the runtime class of this object.

Let us first understand what is runtime class and the object of a runtime class. How it is related to the getClass() method in Java.

* **Runtime class**: The class that is loaded into JVM at execution time is called the runtime class.
* **Runtime class object**: Every class bytecode is stored using java.lang.Class object. This java.lang.Class object is called the runtime class object.

## After loading every .class file, JVM creates one object of java.lang.Class type in the heap area. We can use this object to get class-level information like class name, method information, constructor information, and e.t.c. The getClass() method of the Object class is used very frequently in the reflection API.

**ava Object getClass() Method Example on Pre-defined classes,**

import java.util.ArrayList;

public class Test {

public static void main(String[] args) {

String str = new String();

System.out.println("Class of str: " + str.getClass());

ArrayList al = new ArrayList();

System.out.println("Class of al: " + al.getClass());

}

}

Output:-

Class of str: class java.lang.String

Class of al: class java.util.ArrayList

class A { }

public class Test {

public static void main(String[] args) {

Test t1 = new Test();

System.out.println("Class of t1: " + t1.getClass());

A a1 = new A();

System.out.println("Class of a1: " + a1.getClass());

}

}

Output:-

Class of t1: class Test  
Class of a1: class A

**Java program to get the class name**

class A { }

class Test {

public static void main(String[] args) {

Test t1 = new Test();

String classname = **t1.getClass().getName();**

System.out.println(classname);

A a1 = new A();

classname = **a1.getClass().getName();**

System.out.println(classname);

}

}

Output:-

Test  
A

**public String getName()**

Assume there is a class called “A”. Below code shows retrieving the class name of runtime class object using the getClass() method in Java:-

// create object of class

A a1 = new A();

// get runtime object

Class cls = a1.getClass();

// get class name

String name = **cls.getName();**

The above lines of code can be written in a single line as follows:-

## Using getClass() Method in Java to get Class Information

We can use the runtime class objects to get more information on the class. Some important methods are,

* **getFields()**
* **getConstructors()**
* **getMethods()**

On calling the above methods on the runtime class object, they will give information of the **current class**, and all its **superclass**.

Java program to display all methods of a class using getClass() method of Object class,

import java.lang.reflect.\*;

class Test {

// methods

public void m1(){}

public void m2(){}

public void m3(){}

public static void main(String[] args) {

Test t1 = new Test();

**Class cls = t1.getClass();**

// display class name of runtime class object

String classname = cls.getName();

System.out.println("Class Name: "+ classname);

// display all methods of this Class object

Method[] methods = cls.getMethods();

System.out.println("Methods: ");

for(Method m : methods) {

System.out.print(m.getName() + " ");

}

}

}

Output:-

Class Name: Test  
Methods:  
m3 main m2 m1 wait wait wait equals toString hashCode getClass notify notifyAll

## Different Ways to Get Runtime Class Object in Java

In Java, the getClass() method is not the only way to get the runtime class object. Different ways to get the object of a given class are,

1) Class cls = **classname.class;**  
2) Class cls = **Class.forName(classname);**  
3) Class cls = **obj.getClass();**

Let us take a Test class for demonstration purposes,

public class Test {

public static int i = m1();

static {

System.out.println("Test class: Static block executed");

}

public static void main(String[] args) {

System.out.println("Test class: main method executed");

}

public static int m1() {

System.out.println("Test class: Static variable initialized");

return 5;

}

}

**1) Using classname.class to get the runtime class object,**

// classname.class demonstration

public class Example1 {

public static void main(String[] args) {

Class<Test> t1 = **Test.class;**

System.out.println("Name of class: " + t1.getName());

}

}

Output:-

Name of class: Test

In the first approach, using **Test.class** => Test class is loaded & stored in the JVM, and the runtime class object is returned.

In this approach, the Test class static variables are not initialized, static block and the main method of the Test class are also not executed.

**2) Using Class.forName() to get the runtime class object,**

// Class.forName() demonstration

public class Example2 {

public static void main(String[] args)

throws ClassNotFoundException {

Class<?> t1 = **Class.forName("Test");**

System.out.println("Name of class: "

+ t1.getName());

}

}

Output:-

Test class: Static variable initialized  
Test class: Static block executed  
Name of class: Test

**3) Using getClass() to get the runtime class object,**

// getClass() demonstration

public class Example3 {

public static void main(String[] args){

// creating object of class

System.out.println("Creating an object of Test class");

Test t1 = new Test();

System.out.println("Test class object is created");

// get class reference

Class cls = **t1.getClass();**

System.out.println("getClass() is called");

// get name of class

System.out.println("Name of class: " + cls.getName());

}

}

Output:-

Creating an object of Test class  
Test class: Static variable initialized  
Test class: Static block executed  
Test class object is created  
getClass() is called  
Name of class: Test

In this approach, the already created Class object reference is returned because the Test class has already loaded due to its object creation.

Whenever the constructor is called for the first time (assuming the class was not used before that statement) then the class is loaded, static variables are initialized, and static blocks are executed. But getClass() is not performing these operations.

## Hashcode In Java – HashCode() Method

## What is hashcode in Java?

Hashcode is an identity of an object. In Java, for every object, a unique number is generated by the JVM and it is nothing but a hashcode.

It is a **32-bit unique integer number** and is used to differentiate one object from another object and also used for differentiating one group of objects from other groups of objects.

As per javadocs, the hashcode is defined as “***hashcode is typically implemented by converting the internal address of the object into an integer number***”.

## What is the use of hashcode in Java?

In regular programming, we won’t use the hashcode of an object. JVM uses hashcode value while saving data in hashing-related data structures like HashTable, HashSet, and HashMap. Hashcode is mainly used while working with Set and Map collections for storing, retrieving, removing, and searching objects of a class in the hashtable data structure.

The main advantage of saving data based on hashcode is search, add, and remove operation becomes very easy, which is having time complexity O(1).

The time complexity of different search operations:-  
1) [Linear Search](https://www.knowprogram.com/java/linear-search-in-java/):- O(n)  
2) [Binary Search](https://www.knowprogram.com/java/binary-search-java/):- O(log n)  
3) Hashing:- O(1)

## hashCode() Method of Java Object class

To find hashcode value hashCode() method is defined in [java.lang.Object class](https://www.knowprogram.com/java/object-class-java/). It is a **non-static**, **native method** and its default implementation is returning the reference of the object in integer form.

The hashCode() method signature in java:-  
***public native int hashCode()***  
Return value:- Reference of the object in integer form.

public class Test {

public static void main(String[] args) {

Test t1 = new Test();

Test t2 = new Test();

System.out.println(**t1.hashCode()**);

System.out.println(**t2.hashCode()**);

}

}

Output:-

225534817  
1878246837

Since the reference is varied from one computer to another computer so hashcode value also changed. We may get different hashcode values on the next execution of the program, and you will get different hashcode values on your computer.

Another example to demonstrate the hashCode() method,

class A {

int x;

}

public class Test {

public static void main(String[] args) {

A a1 = new A();

A a2 = new A();

System.out.println(a1.hashCode());

System.out.println(a2.hashCode());

}

}

Output:-

1878246837  
929338653

## Override hashCode() Method in Java

We can also generate a custom hashcode of an object by using its states(values). In this case, we must override the hashCode() method in the subclass by returning the state of the object by developing some hashing algorithm.

class A {

int x;

A(int x){

this.x = x;

}

**@Override**

**public int hashCode() {**

**return x;**

**}**

}

public class Test {

public static void main(String[] args) {

A a1 = new A(10);

A a2 = new A(20);

System.out.println(a1.hashCode());

System.out.println(a2.hashCode());

}

}

Output:-

10  
20

Note:- **Hashcode always doesn’t represent the address of the object**.

If the hashCode() method is not overridden then it will be executed from java.lang.Object class and returns hashCode value based on the object reference. In that case, the hashcode value is representing the address of the object. But if the hashCode() is overridden in the class then it returns a value based on its state/data not based on its reference. Hence in this case hashcode doesn’t represent the address of the object.

public class Student {

int id;

String name;

String course;

public Student(int id, String name, String course) {

this.id = id;

this.name = name;

this.course = course;

}

**@Override**

**public int hashCode() {**

**return (int) id \* name.hashCode() \* course.hashCode();**

**}**

}

public class Test {

public static void main(String[] args) {

Student s1 = new Student(1125, "John", "Java");

System.out.println(s1.hashCode());

Student s2 = new Student(8562, "Sophia", "Python");

System.out.println(s2.hashCode());

}

}

Output:-

1727255470  
827442848

### **Different Cases with Hashcode Value while Overriding hashCode() method**

**1) What will be the hashcode value after changing the value of the object?**

If the hashCode() method is not overridden then after changing the value of an object the hashcode value returned by the hashCode() method won’t change because it is reference-based. But if the hashCode() method is overridden then the hashcode value returned by the hashCode() method will also be changed.

class A {

int x;

A(int x){

this.x = x;

}

@Override

public int hashCode() {

return x;

}

}

public class Test {

int x;

Test(int x) {

this.x = x;

}

public static void main(String[] args) {

// hashCode() is not overriden in Test class

Test t1 = new Test(10);

System.out.println(t1.hashCode());

t1.x = 20; // changing value

System.out.println(t1.hashCode());

// hashCode() is overriden in A class

A a1 = new A(10);

System.out.println(a1.hashCode());

a1.x = 20; // changing value

System.out.println(a1.hashCode());

}

}

Output:-

225534817  
225534817  
10  
20

**2) Two objects can have the same hashcode (returned by hashCode() method)?**

If the hashCode() method is not overridden then two different objects will never have the same hashcode value, because at a time two objects can’t have the same reference. But if the hashCode() method is overridden then its return value may be the same.

class A {

int x;

A(int x){

this.x = x;

}

@Override

public int hashCode() {

return x;

}

}

public class Test {

public static void main(String[] args) {

// hashCode() is not overriden in Test

Test t1 = new Test();

System.out.println(t1.hashCode());

Test t2 = new Test();

System.out.println(t2.hashCode());

// hashCode() is overriden in A class

A a1 = new A(10);

System.out.println(a1.hashCode());

A a2 = new A(10);

System.out.println(a2.hashCode());

}

}

Output:-

225534817  
1878246837  
10  
10

## Alternative Ways to Get JVM Generated hashcode value

JVM always generates a unique number for every object value and hashCode() method of java.lang.Object class is one of the ways to get this value. Every object will have a reference, so every object will have a reference-wise JVM-generated hashcode value.

When we are overriding the hashCode() method in our class then it generates a hashcode value based on state. But the object will also contain the JVM-generated hashcode value based on its reference. How can we get that hashcode value?

If we want to retrieve referenced wise hashcode, we must follow one of the below two approaches:-

1) [**System.identityHashCode()**](https://www.knowprogram.com/java/identityhashcode/):- It is a static native method defined in the System class to return reference-wise JVM generated hashcode of the given object. Its prototype is:- public static native int identityHashCode(Object obj)

2) **super.hashCode()**:- By calling super.hashCode() method from a new method that is defined in the subclass.

Program to demonstrate if we don’t override hashCode() method in our class then hashCode(), System.identityHashCode(), and super.hashCode() returns the same value which is based on the reference of the object.

public class Test {

public int JVMHashCode() {

return super.hashCode();

}

public static void main(String[] args) {

Test t1 = new Test();

System.out.println(t1.hashCode());

System.out.println(System.identityHashCode(t1));

System.out.println(t1.JVMHashCode());

Test t2 = new Test();

System.out.println(t2.hashCode());

System.out.println(System.identityHashCode(t2));

System.out.println(t2.JVMHashCode());

}

}

Output:-

225534817  
225534817  
225534817  
1878246837  
1878246837  
1878246837

If we override hashCode() method then the hashCode() method return value based on state. And we can use System.identityHashCode(), or super.hashCode() method to get the JVM generated hashcode value.

class A {

int x;

A(int x){

this.x = x;

}

@Override

public int hashCode() {

return x;

}

public int JVMHashCode() {

return super.hashCode();

}

}

public class Test {

public static void main(String[] args) {

A a1 = new A(10);

System.out.println(a1.hashCode());

System.out.println(System.identityHashCode(a1));

System.out.println(a1.JVMHashCode());

A a2 = new A(20);

System.out.println(a2.hashCode());

System.out.println(System.identityHashCode(a2));

System.out.println(a2.JVMHashCode());

}

}

Output:-

10  
1878246837  
1878246837  
20  
929338653  
929338653

## hashCode() vs [identityHashCode()](https://www.knowprogram.com/java/identityhashcode/)

The hashCode() method can be overridden but System.identityHashCode() can’t be overridden. The hashCode() method is a non-static method so, to call this method object should be there. The System.identityHashCode() is static method. Therefore It also can be used to get the hashcode value of the primitive variables.

public class Test {

public static void main(String[] args) {

int x = 10;

System.out.println(System.identityHashCode(x));

// System.out.print(x.hashCode()); // error

// error: int cannot be dereferenced

}

}

Output:-

648436640

## Object Class ToString() Method In Java

**Object Class toString() Method in Java** | The java.lang.Object.toString() method is used to retrieve object information in string form. The prototype of the toString() method in the [Object class in Java](https://www.knowprogram.com/java/object-class-java/) is:- **public String toString()**

Every class is having its own class name, hashcode, and state. In java.lang.Object class toString() method is implemented for returning the class name and hashcode of the current object in the hexadecimal format as classname@hashCodeInHexadecimalFormat. It is internally calling [**getClass() method**](https://www.knowprogram.com/java/getclass-method-java/) and [**hashCode() method**](https://www.knowprogram.com/java/hashcode-method/). The toString method implementation logic is given in the [Object class](https://www.knowprogram.com/java/object-class-java/) as shown below:

public String toString() {

return getClass().getName() + "@"

+ Integer.toHexString(hashCode());

}

Let us demonstrate it through an example,

public class Test {

public static void main(String[] args) {

Test t1 = new Test();

System.out.println(t1);

System.out.println(t1.toString());

Test t2 = new Test();

System.out.println(t2);

}

}

Output:-

Test@d716361  
Test@d716361  
Test@6ff3c5b5

Whenever we are trying to print an Object reference then internally toString() method will be called. Since the toString() method is not defined in the Test class, so the compiler will search for it in the superclass. Directly or indirectly every Java class is a subclass of java.lang.Object class. Therefore, the toString() method will be executed from the Object class and display the string “classname@hashCodeInHexadecimalFormat”.

Note that the below two lines give the same result. Whenever we call print()/println() method to display object then internally it called toString() method.

System.out.println(t1); // internally toString() is called

System.out.println(t1.toString());

Another example with Student class,

class Student {

int id;

String name;

String course;

public Student(int id, String name, String course) {

this.id = id;

this.name = name;

this.course = course;

}

}

public class Test {

public static void main(String[] args) {

Student s1 = new Student(100, "Emma", "Java");

System.out.println(s1);

Student s2 = new Student(101, "Noah", "Python");

System.out.println(s2);

}

}

Output:-

Student@ee7d9f1  
Student@1963006a

[Java toString() Quiz](https://www.knowprogram.com/java/java-tostring-quiz/)

Override toString() in Java

The above outputs are not meaningful messages. It should display Student object data, rather than its references. If we want to return the object’s state from this method we must override it in the subclass. Based on our requirement we can override the toString() method in our class to provide our own representation.

public class Test {

@Override

public String toString() {

return "Test";

}

public static void main(String[] args) {

Test t1 = new Test();

System.out.println(t1);

System.out.println(t1.toString());

Test t2 = new Test();

System.out.println(t2);

}

}

Output:-

Test  
Test  
Test

Now, in the Test class toString() method is overridden and returns the “Test” string. Therefore whenever we try to print object reference then it is executed from the Test class and returns the “Test” String. Let us see another example:-

class Student {

int id;

String name;

String course;

public Student(int id, String name, String course) {

this.id = id;

this.name = name;

this.course = course;

}

@Override

public String toString() {

return "Student [id=" + id +

", name=" + name +

", course=" + course + "]";

}

}

public class Test {

public static void main(String[] args) {

Student s1 = new Student(100, "Emma", "Java");

System.out.println(s1);

Student s2 = new Student(101, "Noah", "Python");

System.out.println(s2);

}

}

Output:-

Student [id=100, name=Emma, course=Java]  
Student [id=101, name=Noah, course=Python]

In the Student class toString() method is overridden and returns the state or data of the object. Therefore whenever we are trying to print an object reference then it will display the id and name of the Student object.

Use of toString Method in Java

**In all**[**collection classes**](https://www.knowprogram.com/java/java-collection-framework-overview/)**, all wrapper classes,**[**Arrays**](https://www.knowprogram.com/java/arrays-class-java/)**, String, StringBuffer, and StringBuilder classes the toString() method is overridden to display the content.** That’s why while using these classes we are getting object data rather than the reference of the object.

import java.util.ArrayList;

import java.util.Arrays;

public class Test {

public static void main(String[] args) {

// String class

String s = "KnowProgram";

System.out.println(s);

// Wrapper classes

Integer io = Integer.valueOf(10);

System.out.println(io);

Double d = Double.valueOf(19.5);

System.out.println(d);

// ArrayList class

ArrayList<Character> al = new ArrayList<Character>();

al.add('K');

al.add('P');

System.out.println(al);

// Arrays

int arr[] = new int[]{1, 2, 3, 4, 5};

System.out.println(Arrays.toString(arr));

}

}

Output:-

KnowProgram  
10  
19.5  
[K, P]  
[1, 2, 3, 4, 5]

See More:- [Arrays.toString() in Java](https://www.knowprogram.com/java/arrays-to-string/)

Relation between toString() and [hashCode()](https://www.knowprogram.com/java/hashcode-method/) methods

When the toString() method is not overridden in the class then the toString() method is executed from the [Java Object class](https://www.knowprogram.com/java/object-class-java/). **Object class toString() method internally calls hashCode() method.**

But if we are overriding the toString() method then it may or may not call hashCode(), it completely depends on the implementation logic.

**#Case1:- Both toString() and hashCode() methods are not overriden.**

class A{

int x;

A(int x){

this.x = x;

}

}

class Test{

public static void main(String[] args) {

A a1 = new A(18);

System.out.println(a1);

}

}

Output:-

A@6ff3c5b5

Here toString() and hashCode() methods are executed from Object class and returns className@ hashCodeInHexadecimalFormat

**#Case2:- Only hashCode() method is overriden.**

class A{

int x;

A(int x){

this.x = x;

}

@Override

public int hashCode(){

return 100;

}

}

class Test{

public static void main(String[] args) {

A a1 = new A(18);

System.out.println(a1);

}

}

Output:-

A@64

Whenever we try to print object reference then the toString() method is called. Since toString() is not overridden in the “A” class so it will be called from the Object class. The toString() of the Object class internally calls the hashCode() method, but the hashCode method is overridden in class A. So, the hashCode() method will be executed from the “A” class.

The value returned by the hashCode() method is in integer format which will be converted to hexadecimal format by the toString() method of the Object class. After that, this value will be concatenated with the String. In our example, hashCode() returned 100. The hexadecimal format of integer value 100 is 64. Hence we get A@64.

**#Case3:- The hashCode() and toString() method, both are overriden.**

class A{

int x;

A(int x){

this.x = x;

}

@Override

public int hashCode(){

return 9;

}

@Override

public String toString() {

return "" + x;

}

}

class Test{

public static void main(String[] args) {

A a1 = new A(18);

System.out.println(a1);

}

}

Output:-

18

Since toString() method is overridden and its implementation logic is not calling any other methods, directly returns the value in string format. Hence hashCode() method won’t get any chance to execute.

**#Case4:- Only toString() is overriden.**

It is similar to the previous case. When toString() is overridden then it may or may not call hashCode(), it depends on its implementation logic.

## equals() Method in Java Object Class

**equals() Method in Java Object Class** | We can use the equals() method in Java to check the equality of two objects. The equals() method is given to compare two objects of a class for their equality based on their reference (default implementation) or based on data (after overriding).

The equals() method is defined in [java.lang.Object class](https://www.knowprogram.com/java/object-class-java/) and **compare two objects based on their reference**. If both have the same reference then it returns true else it returns false.

The implementation of the equals() method in java.lang.Object class is:-

public boolean equals(Object obj) {

return (this == obj);

}

Java equals() Method Example

Examples to demonstrate Object class equals() method in Java,

public class Test {

public static void main(String[] args) {

Test t1 = new Test();

Test t2 = new Test();

Test t3 = t1;

System.out.println(t1.equals(t2));

System.out.println(t1.equals(t3));

System.out.println(t2.equals(t3));

}

}

Output:-

false  
true  
false

The t1 and t2 are two different objects of the Test class, t3 is pointing to the reference of object t1. Hence, t1 and t2 have different references but t3 and t1 are having the same reference.

The equals() method is called on these objects. Since the equals() method is not available in the Test class so, the compiler searches for it in the superclass. Indirectly or directly every valid Java class is the subclass of java.lang.Object class. Hence equals() method will be executed from the Object class which contains logic to compare two objects based on their reference.

Example of Employee class to demonstrate the Object class equals() method in Java,

class Employee {

int id;

String name;

Employee(int id, String name) {

this.id = id;

this.name = name;

}

}

public class Test {

public static void main(String[] args) {

// create objects

Employee e1 = new Employee(101, "Emma");

Employee e2 = new Employee(102, "Oliver");

Employee e3 = e1;

Employee e4 = new Employee(101, "Emma");

// display details

System.out.println(e1.id+" "+ e1.name);

System.out.println(e2.id+" "+ e2.name);

System.out.println(e3.id+" "+ e3.name);

System.out.println(e4.id+" "+ e4.name);

// compare objects

System.out.println(e1.equals(e2)); // false

System.out.println(e1.equals(e3)); // true

System.out.println(e1.equals(e4)); // false

}

}

Output:-

101 Emma  
102 Oliver  
101 Emma  
101 Emma  
false  
true  
false

If two references are pointing to the object then only the equals() method of java.lang.Object class returns true, otherwise, it returns false.

In the above example, e1 and e4 contain exactly the same data. Since they have different references and the Object class equals() method compares objects based on reference, therefore, it gives the result as false.

Override equals() Method in Java

If we want to compare two objects with state or value then we must override the equals() method in the subclass. Many times objects are created and passed as method arguments at runtime. So, to know whether the given object are same or not, we must compare objects using their state.

**Java Program to compare two objects by overriding equals() method**

class Employee {

int id;

String name;

Employee(int id, String name) {

this.id = id;

this.name = name;

}

**@Override**

**public boolean equals(Object obj) {**

**if(this == obj){**

**return true;**

**}**

**if(obj instanceof Employee) {**

**Employee e = (Employee) obj;**

**return (this.id == e.id) &&**

**this.name.equals(e.name);**

**}**

**return false;**

**}**

}

public class Test {

public static void main(String[] args) {

// create objects

Employee e1 = new Employee(101, "Emma");

Employee e2 = new Employee(102, "Oliver");

Employee e3 = e1;

Employee e4 = new Employee(101, "Emma");

// compare objects

System.out.println(e1.equals(e2)); // false

System.out.println(e1.equals(e3)); // true

System.out.println(e1.equals(e4)); // true

}

}

Output:-

false  
true  
true

How to Override equals() Method in Java

**Step-1**) While overriding first **check the passed object and current object are pointing to the same reference**. If both are pointing to the same reference then both objects are always the same and they will contain the same data.

For this, write below condition in equals method as,

// compare passed object and current

// object reference

**if(this == obj){**

**return true;**

**}**

For example- e1 and e3 are two different objects but they are pointing to the same reference, then it will return true.

Employee e1 = new Employee(101, "Emma");

Employee e3 = e1;

System.out.println(e1.equals(e3)); // true

**Step-2**) Before comparing the values of the object, **compare whether both objects are compatible or not**. For this, we can use instanceof operator.

**if(obj instanceof Employee) {**

// logic to compare data of objects

**} else {**

**return false;**

**}**

It will handle situations like comparing with null, or incompatible type objects. Examples:-

Employee e1 = new Employee(101, "Emma");

System.out.println(e1.equals(null)); // false

Test t1 = new Test();

System.out.println(e1.equals(t1)); // false

If we try to compare Employee type objects with Student, Test, or some other type object then it must return false because they are not compatible with each other.

**Step-3) Convert object type to current class type.**

The parameter of the equals() method is an Object type. To compare it with another object, it should be converted to the current class type.

**Employee e = (Employee) obj;**

**Step-4**) Finally **compare all values** of the object.

Here, to compare two primitive values use the == operator, and to compare objects use the equals() method. For example, their id and name are two values in the Employee class. Since id is a primitive variable so we will use the == operator and the name is of String type hence equals() method will be used.

// get current object values

int id1 = this.id;

String name1 = this.name;

// get passed object values

int id2 = e.id;

String name2 = e.name;

// compare values

if((id1 == id2) && (name1.equals(name2)) {

return true;

}

The current object can be fetched directly by its name or using the “this” keyword. We can reduce the above lines as,

**return (this.id == e.id) && this.name.equals(e.name);**

Or,

return (id == e.id) && name.equals(e.name);

Uses of the “this” keyword improve the readability of the code therefore it is recommended to use the “this” keyword while comparing the data of the objects.

Uses of equals() Method in Java

In all wrapper classes, the [Arrays class](https://www.knowprogram.com/java/arrays-class-java/), and String class equals() method is overridden to compare the data of the objects. But in the StringBuffer and StringBuilder class, the equals() method are not overridden, they will use the Object class equals() method to compare the objects. Example:-

public class Test {

public static void main(String[] args) {

String str1 = new String("Know Program");

String str2 = new String("Know Program");

System.out.println(str1.equals(str2));

StringBuilder sb1 = new StringBuilder("Know Program");

StringBuilder sb2 = new StringBuilder("Know Program");

System.out.println(sb1.equals(sb2));

}

}

Output:-

true  
false

The Set collection classes allow storing only unique elements, they don’t allow storing duplicate elements. The duplicate elements are found with the help of the equals() method. If the equals() method return true then they are duplicate elements.

Example-2 to Override equals method in Java

Write code to override the equals() method in the given Student class.

class Student {

private int rollno;

private String name;

private String dept;

private double avg;

// constructor

Student(int rollno, String name,

String dept, double avg) {

this.rollno = rollno;

this.name = name;

this.dept = dept;

this.avg = avg;

}

}

Use == operator to compare primitive values, and use equals() method to compare Objects.

@Override

public boolean equals(Object obj) {

if(this == obj) return true;

if(obj instanceof Student) {

Student s = (Student) obj;

return

( (this.rollno == s.rollno) &&

this.name.equals(s.name) &&

this.dept.equals(s.dept) &&

(this.avg == s.avg) );

}

return false;

}

Example-3 to Override equals method in Java

We have 2 classes Student and Address. Each Student has their id, name, and Address. Each Address contains the house number and city. The main task is to check two students are the same or not?

2 Students are the same if they have exactly the same id, name, and Address. In Adress, they must have the same house number and city. Therefore we have to override the equals() method in both Student and Address classes.

Address class with fields hno (house name), and city,

class Address {

int hno; // house-number

String city;

Address(int hno, String city) {

this.hno = hno;

this.city = city;

}

@Override

public boolean equals(Object obj) {

if(this == obj) return true;

if(obj instanceof Address) {

Address ad = (Address)obj;

return (this.hno == ad.hno) &&

this.city.equals(ad.city);

}

return false;

}

}

Student class with fields id, name, and adrs(Address),

class Student {

int id;

String name;

Address adrs;

public Student(int id, String name, Address adrs) {

this.id = id;

this.name = name;

this.adrs = adrs;

}

@Override

public boolean equals(Object obj) {

if(this == obj) return true;

if(obj instanceof Student) {

Student st = (Student) obj;

return (this.id == st.id) &&

(this.name.equals(st.name)) &&

(this.adrs.equals(st.adrs));

}

return false;

}

}

Test class to compare students,

public class Test {

public static void main(String[] args) {

Student st1 =

new Student(101, "Sophia", new Address(8546, "Los Angeles"));

Student st2 =

new Student(101, "Sophia", new Address(8546, "Los Angeles"));

Student st3 =

new Student(101, "Sophia", new Address(9853, "San Francisco"));

System.out.println(st1.equals(st2)); // true

System.out.println(st1.equals(st3)); // false

}

}

Output:-

true  
false

If you want to display Student data then you have to override the toString() method. See more:- [toString() method in Java Object class](https://www.knowprogram.com/java/object-tostring-java/).

Object Comparison in Java with equals() Method

We can use the equals method to compare different types of objects. We have already seen the comparison of two different objects.

|  |  |  |
| --- | --- | --- |
| **Comparison using equals()** | **Valid?** | **Return** |
| Two objects | Yes | false/true |
| Two primitive variables | No | compile-time error |
| Two nulls | No | compile-time error |
| Null with object | No | compile-time error |
| Object with null | Yes | false |
| Null referenced with null | No | NullPointerException |
| Null referenced with object | No | NullPointerException |
| Two Null referenced | No | NullPointerException |
| Object with null referenced | Yes | false |
| Incompatible objects | Yes | false |

We can’t use the equals() method on primitive values because equals() is an instance method and to call equals() method object should be there.

int i1 = 10;

int i2 = 20;

System.out.println(i1.equals(i2)); // error

We can’t compare two nulls, or nulls with objects because a null is also treated as a primitive type value.

System.out.println(null.equals(null)); // error

Test t1 = new Test();

System.out.println(null.equals(t1)); // error

Comparing Object with null always gives “false” as a result.

Test t1 = new Test();

System.out.println(t1.equals(null)); // false

Using the equals() method on null referenced with null, null referenced with an object, and two nulls referenced always raise NullPointerException at runtime.

Test t1 = null;

System.out.println(t1.equals(null)); // Exception

Test t2 = new Test();

System.out.println(t1.equals(t2)); // Exception

Test t3 = null;

System.out.println(t1.equals(t3)); // Exception

Using the equals() method on Object with null referenced, and Incompatible objects always gives false as result.

Test t1 = new Test();

Test t2 = null;

System.out.println(t1.equals(t2)); // false

Student st1 = new Student();

System.out.println(t1.equals(st1)); // false

## hashCode() and equals() Method in Java

**HashCode() and equals() method in Java** | Previously we discussed [hashCode() method](https://www.knowprogram.com/java/hashcode-method/) and [equals() method](https://www.knowprogram.com/java/equals-method-java/) of [java.lang.Object class](https://www.knowprogram.com/java/object-class-java/) in detail. Now let us see the relation between hashCode() and the equals() method in Java. What are the differences between the hashCode() and equals() methods in Java?

What is the contract between the hashcode and the equals method?

There is a contract between the equals() and hashCode() method in Java, if two objects are equal by the equals() method then their hashCode() must be equal i.e. **two equivalent objects should have the same hashcode**.

class Student {

int id;

Student(int id) {

this.id = id;

}

}

Assume s1 and s2 are two different objects of the Student class. **If s1.equals(s2) is true then s1.hashCode() == s2.hashCode() is always true.**

[java.lang.Object class](https://www.knowprogram.com/java/object-class-java/) equals() and hashCode() method follows above contract. Hence whenever we are overriding the equals() method then compulsory we should override the hashCode() method to satisfy the above contract i.e. two equivalent objects should have the same hashCode. If we don’t follow this contract then we won’t get any compile-time error, or runtime exception but it is not a good programming practice.

In java.lang.String class equals() method is overridden for content comparison and hence hashCode() method is also overridden to generate hashcode value based on the content.

public class Test {

public static void main(String[] args) {

String s1 = new String("KnowProgram");

String s2 = new String("KnowProgram");

System.out.println(s1.equals(s2));

System.out.println(s1.hashCode() == s2.hashCode());

System.out.println(s1.hashCode());

System.out.println(s2.hashCode());

}

}

Output:-

true  
true  
2058267993  
2058267993

In StringBuffer and StringBuilder classes, the equals() method is not overridden for content comparison, therefore the hashCode() method is also not overridden.

public class Test {

public static void main(String[] args) {

StringBuilder s1 = new StringBuilder("KnowProgram");

StringBuilder s2 = new StringBuilder("KnowProgram");

System.out.println(s1.equals(s2));

System.out.println(s1.hashCode() == s2.hashCode());

System.out.println(s1.hashCode());

System.out.println(s2.hashCode());

}

}

Output:-

false  
false  
225534817  
1878246837

Relation between hashCode() and equals() Method in Java

1) If two objects are not equal by the equals() method then their hashcode value may or may not be the same.

|  |  |
| --- | --- |
| **s1.equals(s2)** | **s1.hashCode() == s2.hashCode()** |
| true | true |
| false | true/false |

2) If the hashcode of two objects is equal then the equals() method return true or false.  
3) If the hashcode of two objects is not equal then these objects are always not equal by the equals() method.

|  |  |
| --- | --- |
| **s1.hashCode() == s2.hashCode()** | **s1.equals(s2)** |
| true | true/false |
| false | false |

Overriding hashCode() and equals() Method in Java

**Based on which parameter we override the equals() method, it is highly recommended to use the same parameter while overriding hashCode() also.**

In all collections classes, wrapper classes, and string classes the equals() method is overridden for content comparison. Hence it is recommended to override the equals() method in our class for content comparison.

In the [equals() method](https://www.knowprogram.com/java/equals-method-java/) we have already discussed how to override equals() and the steps to be followed. Now, we will use that knowledge to develop the Student class.

class Student {

private int id;

Student(int id) {

this.id = id;

}

@Override

public boolean equals(Object obj) {

if(this == obj) return true;

if(obj instanceof Student) {

Student s = (Student) obj;

return this.id == s.id;

}

return false;

}

@Override

public int hashCode() {

return id;

}

}

public class Test {

public static void main(String[] args) {

Student s1 = new Student(1001);

Student s2 = new Student(1002);

Student s3 = new Student(1001);

// invoke hashCode() method

System.out.println("hashCode():: " +

s1.hashCode() + " " +

s2.hashCode() + " " +

s3.hashCode() );

// invoke equals() method

System.out.println("equals():: " +

s1.equals(s2) + " " +

s1.equals(s3) + " " +

s2.equals(s3) );

}

}

Output:-

hashCode():: 1001 1002 1001  
equals():: false true false

## Clone() Method in Java

**Clone() Method in Java Object class** | Cloning of objects means creating an exact duplicate copy with the current object state. In Java, to perform a cloning **clone()**method is given in [java.lang.Object class](https://www.knowprogram.com/java/object-class-java/).

The prototype of java.lang.Object.clone() method is:-  
**protected native Object clone() throws CloneNotSupportException**

Condition:- To execute clone() method on an object the **class must implements java.lang.Cloneable interface**, else this method throws exception “java.lang.CloneNotSupportException”

We can perform cloning only on clonable objects. An object is said to be clonable if and only if the corresponding class implements the Cloneable interface. Cloneable is a marker interface present in java.lang package. It provides permission to execute the clone() method to clone the object.

## Need of Cloning

The need of cloning or the main purpose of cloning is,

* **To maintain a backup copy.**
* **To preserve the state of an object.**

Whenever we are assigning one object reference to another object then after changing one object state, another object state also changed.

class Student {

int id;

Student(int id) {

this.id = id;

}

}

class Test {

public static void main(String[] args) {

Student s1 = new Student(9);

Student s2 = s1;

System.out.println(s1 == s2);

// display details

System.out.println(s1.id+" "+s2.id);

// change original object state

s1.id = 100;

// state after change

System.out.println(s1.id+" "+s2.id);

}

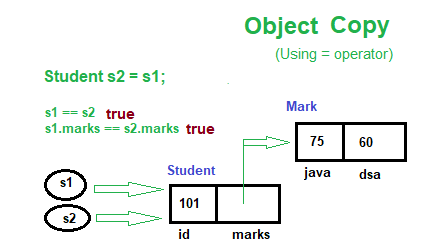
}

Output:-

true  
9 9  
100 100

Here s1 and s2 point to the same reference. Therefore after changing the id of s1, modification is also reflected in the s2.

We can notice that after changing one object state another object state also changed. To solve this problem, we need to go for cloning operations. The below image shows how the = operator just assigns the reference of the original object to the duplicate object.



## Object Class clone() Method

Rules to invoke clone() method of Object class,

1. To execute the clone() method the current **object should be Cloneable** type else it leads to the exception CloneNotSupportedException.
2. It is a **protected method**, therefore it can be called on a class object only inside that class. If we call it in other classes including in subclass it leads to a compile-time error. To call it from other classes we must override the clone() method in that subclass with the public keyword.
3. The **return type** of clone() method is **java.lang.Object type**. Therefore, we must cast the clone method returning the object to its current object class.
4. The clone() method throws **CloneNotSupportedException** which is a checked exception. We must handle it either by catching it using a try/catch block or by reporting it using the throws keyword.

class Student implements Cloneable {

int id;

Student(int id) {

this.id = id;

}

public static void main(String[] args)

throws CloneNotSupportedException {

// create objects

Student s1 = new Student(9);

Student s2 = (Student) s1.clone();

// compare references

System.out.println(s1 == s2);

// display details

System.out.println(s1.id+" "+s2.id);

// change original object state

s1.id = 100;

// state after change

System.out.println(s1.id+" "+s2.id);

}

}

Output:-

false  
9 9  
100 9

After changing one object state another object state doesn’t change. After modifying the id of s1 modification doesn’t affect s2.

We invoked clone() within a subclass, if we try to invoke the clone() method outside of this class (like Test class) then we will get a **compile-time error: clone() has protected access in Object**. To solve this problem we must override the clone() method.

[clone() Method Quiz](https://www.knowprogram.com/java/clone-method-quiz/)

## Overriding clone() method in Java

If we want to call the clone() method from its user class then we must override the clone() method.

Generally, we override methods for two reasons,  
1) To change implementation logic. Examples:- Overriding toString(), hashCode(), and equals() method.  
2) To change the accessibility modifiers. Example:- Overriding clone() method.

Procedure to override clone() method,

* We are overriding the clone() method just to change the accessibility, not to change the implementation logic. Therefore we must call the object class clone() method while overriding as super.clone().
* The Object class clone() method return type is java.lang.Object, but while overriding we can implement covariant returns (Student type). So that the user/test class developer no need to downcast the cloned object. Note:- But before returning we must downcast the return value.

@Override

public Student clone() throws CloneNotSupportedException {

// downcast the return value

return (Student) super.clone();

}

Now let us see the example of overriding the clone() method in Java,

class Student implements Cloneable {

int id;

Student(int id) {

this.id = id;

}

@Override

public Student clone()

throws CloneNotSupportedException {

return (Student) super.clone();

}

}

class Test {

public static void main(String[] args)

throws CloneNotSupportedException {

// create objects

Student s1 = new Student(9);

Student s2 = s1.clone();

// compare references

System.out.println(s1 == s2);

// display details

System.out.println(s1.id+" "+s2.id);

// change original object state

s1.id = 100;

// state after change

System.out.println(s1.id+" "+s2.id);

}

}

Output:-

false  
9 9  
100 9

After changing the value of the primitive variable of the original object, the new object value doesn’t change.

## Cloning Java Object with HAS-A relation

Java supports two types of cloning,  
1) Shallow cloning  
2) Deep cloning

### **Shallow Cloning in Java**

The process of creating a bitwise copy of an object is called shallow cloning. **If the main object contains primitive variables then exactly duplicate copies will be created in the cloned object. If the main object contains any reference variable then the corresponding object won’t be created, just the reference variable will be created pointing to the old object.**

Note:- The Object class clone() method is implemented to perform shallow cloning.

Java Shallow Cloning example,

class Mark {

int java;

int dsa;

Mark(int java, int dsa) {

this.java = java;

this.dsa = dsa;

}

}

class Student implements Cloneable {

int id;

Mark marks;

Student(int id, Mark marks) {

this.id = id;

this.marks = marks;

}

@Override

public Student clone()

throws CloneNotSupportedException {

return (Student) super.clone();

}

}

class Test {

public static void main(String[] args)

throws CloneNotSupportedException {

// create objects

Student s1 = new Student(101, new Mark(75, 60));

Student s2 = s1.clone();

System.out.println(s1 == s2);

System.out.println(s1.marks == s2.marks);

// Java mark of original and new object

System.out.println(s1.marks.java+" "+s2.marks.java);

// changing Java marks of original object

s1.marks.java = 90;

// After change

System.out.println("After change,");

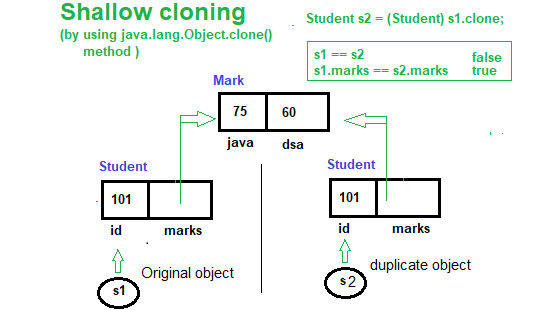
System.out.println(s1.marks.java+" "+s2.marks.java);

}

}

Output:-

false  
true  
75 75  
After change,  
90 90



In shallow cloning, by using cloned object reference if we perform any change to contain values then those changes will be reflected the new object. To overcome this problem we should go for deep cloning.

### **Deep Cloning in Java**

The process of creating an exact duplicate independent copy including the content of the object is called deep cloning.

In deep cloning, if the main object contains any primitive variable then in the cloned object duplicate copies will be created. If the main object contains any reference variable then the corresponding content object also will be created in the cloned copy.

Deep clone copies all the levels of the object from top to bottom recursively. The developer must develop deep cloning by overriding the clone() method.

Procedure:-  
1) Drive both Student and Mark classes from the Cloneable interface.  
2) Override clone() method in both Student and Mark class as public.  
3) Call the Mark class clone() method on this.marks object in the Student class clone() method.

Deep Cloning in Java Example,

class Mark implements Cloneable {

int java;

int dsa;

Mark(int java, int dsa) {

this.java = java;

this.dsa = dsa;

}

@Override

public Mark clone()

throws CloneNotSupportedException {

return (Mark) super.clone();

}

}

class Student implements Cloneable {

int id;

Mark marks;

Student(int id, Mark marks) {

this.id = id;

this.marks = marks;

}

@Override

public Student clone()

throws CloneNotSupportedException {

Student s = (Student) super.clone();

s.marks = this.marks.clone();

return s;

}

}

class Test {

public static void main(String[] args)

throws CloneNotSupportedException {

// create objects

Student s1 = new Student(101, new Mark(75, 60));

Student s2 = s1.clone();

System.out.println(s1 == s2);

System.out.println(s1.marks == s2.marks);

// Java mark of original and new object

System.out.println(s1.marks.java+" "+s2.marks.java);

// changing Java marks of original object

s1.marks.java = 90;

// After change

System.out.println("After change,");

System.out.println(s1.marks.java+" "+s2.marks.java);

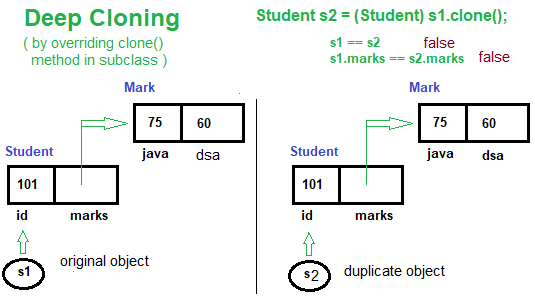
}

}

Output:-

false  
false  
75 75  
After change,  
90 75

Now, using cloned object reference if we perform any change to the content object then those changes won’t be reflected in the new object.



**Note**:- If the object contains only primitive variables then shallow cloning (default implementation of clone() method in Object class) is the best choice. But if the object contains a reference variable then the deep cloning (override clone() method) is the best.

## Cloning object with IS-A relation

The clone() method of the Object class clones the object inheritance graphs starting from the root superclass (java.lang.Object class) to the current cloning subclass. Therefore when an object is cloned then its superclass non-static variables are also cloned. And here, the **only subclass should be of type Cloneable**, the superclass need not be of type Cloneable.

In other words (indirect way) we can say, for IS-A relation Java supports only deep cloning.

class GrandFather {

int house = 2;

}

class Father extends GrandFather {

int money = 500\_000;

}

class Child extends Father implements Cloneable {

int car = 3;

@Override

public String toString() {

return "" + house +" " + money + " " + car;

}

public static void main(String[] args)

throws CloneNotSupportedException{

// objects

Child c1 = new Child();

Child c2 = (Child) c1.clone();

// display data

System.out.println("c1 object details: ");

System.out.println(c1);

System.out.println("c2 object details: ");

System.out.println(c2);

// modifing values

c1.house = 10;

c1.money = 5000;

c1.car = 1;

System.out.println("After change,");

// display data

System.out.println("c1 object details: ");

System.out.println(c1);

System.out.println("c2 object details: ");

System.out.println(c2);

}

}

Output:-

c1 object details:  
2 500000 3  
c2 object details:  
2 500000 3  
After change,  
c1 object details:  
10 5000 1  
c2 object details:  
2 500000 3

If the superclass is having an internal object then that internal object is not cloned only referenced variable memory is cloned.

class A { }

class B { }

class C {

A a1 = new A();

}

class D extends C implements Cloneable {

B b1 = new B();

public static void main(String[] args)

throws CloneNotSupportedException {

// objects

D d1 = new D();

D d2 = (D) d1.clone();

System.out.println(d1.a1 == d2.a1); // true

System.out.println(d1.b1 == d2.b1); // true

d1.a1 = new A();

d1.b1 = new B();

System.out.println(d1.a1 == d2.a1); // false

System.out.println(d1.b1 == d2.b1); // false

}

}