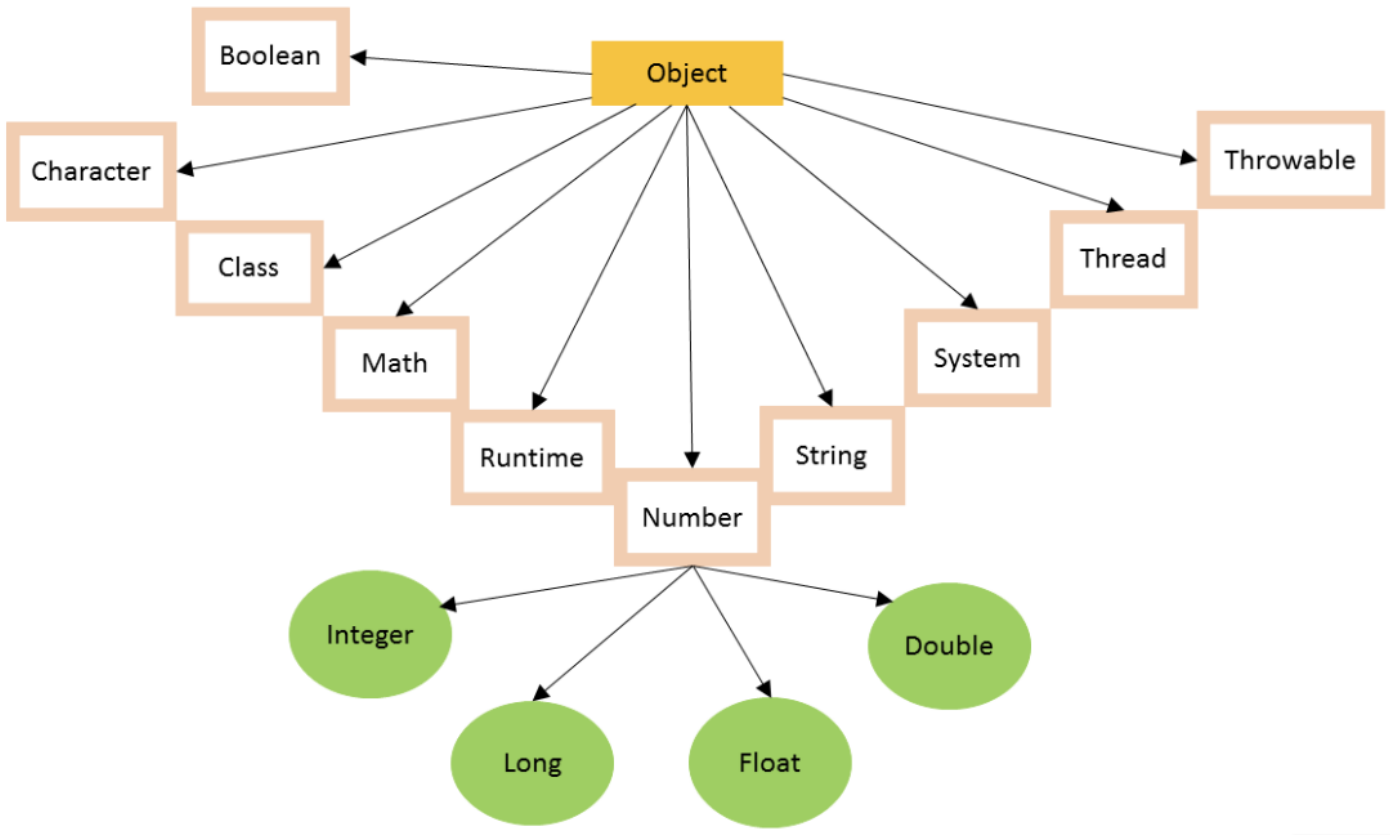
# 15-Topmost Class Object

*Author: Vincent Lau*

## Introduction



In object-oriented programming, a "superclass" (also known as a "base class" or "parent class") is a class from which other classes can inherit properties and behaviors. **It is at the top of the class hierarchy and serves as a blueprint for its subclasses**. In Java, the `Object` class is considered the ultimate superclass for all other classes. Let's dive into more details:

## Class Hierarchy

In Java, classes can be organized into a hierarchy, where each class can have one direct superclass, but potentially multiple subclasses. A subclass inherits the attributes (fields) and behaviors (methods) of its superclass. It can also have additional fields and methods specific to itself.

```
          Object
            ^
            |
          Animal (hypothetical superclass)
            ^
            |
         Mammal (subclass of Animal)
```

In this example, the `Object` class is at the very top of the hierarchy, and all other classes in Java are either direct or indirect subclasses of `Object`.

## Inheritance

Inheritance is the mechanism that allows a class (subclass) to acquire properties and behaviors from its superclass. The subclass keyword `extends` is used in Java to declare that a class inherits from another class (its superclass). When a subclass inherits from a superclass, it gains access to all public and protected members of the superclass (except constructors).

Example:

```java
class Animal {
    protected String name;

    public void makeSound() {
        System.out.println("Some generic animal sound");
    }
}

class Dog extends Animal {
    public Dog(String name) {
        this.name = name;
    }

    @Override
    public void makeSound() {
        System.out.println("Dog sound!");
    }

    public static void main(String[] args) {
        Dog dog = new Dog("ABC");
        dog.makeSound(); // "Dog sound!"
    }
}
```

Here, the `Dog` class is a subclass of `Animal`, and it inherits the `name` field and the `makeSound()` method from the `Animal` class.

## Ultimate Superclass

In Java, every class that you create directly or indirectly (through inheritance) is a subclass of the `Object` class. This means that the `Object` class acts as the ultimate superclass in the Java class hierarchy. Since all classes inherit from `Object`, they **inherit all of its fundamental methods**, such as `toString()`, `equals()`, and `hashCode()`, which **can be overridden** to provide custom behavior for specific classes.

Example:

```java
class MyClass { // This class implicitly extends Object.
    // Fields and methods specific to MyClass.
}
```

Because `MyClass` does not explicitly extend any other class, it implicitly inherits from `Object`.

### Direct vs. Indirect Subclasses

A class can be either a direct subclass (immediate child) or an indirect subclass (descendant) of another class. For instance, consider the following hierarchy:

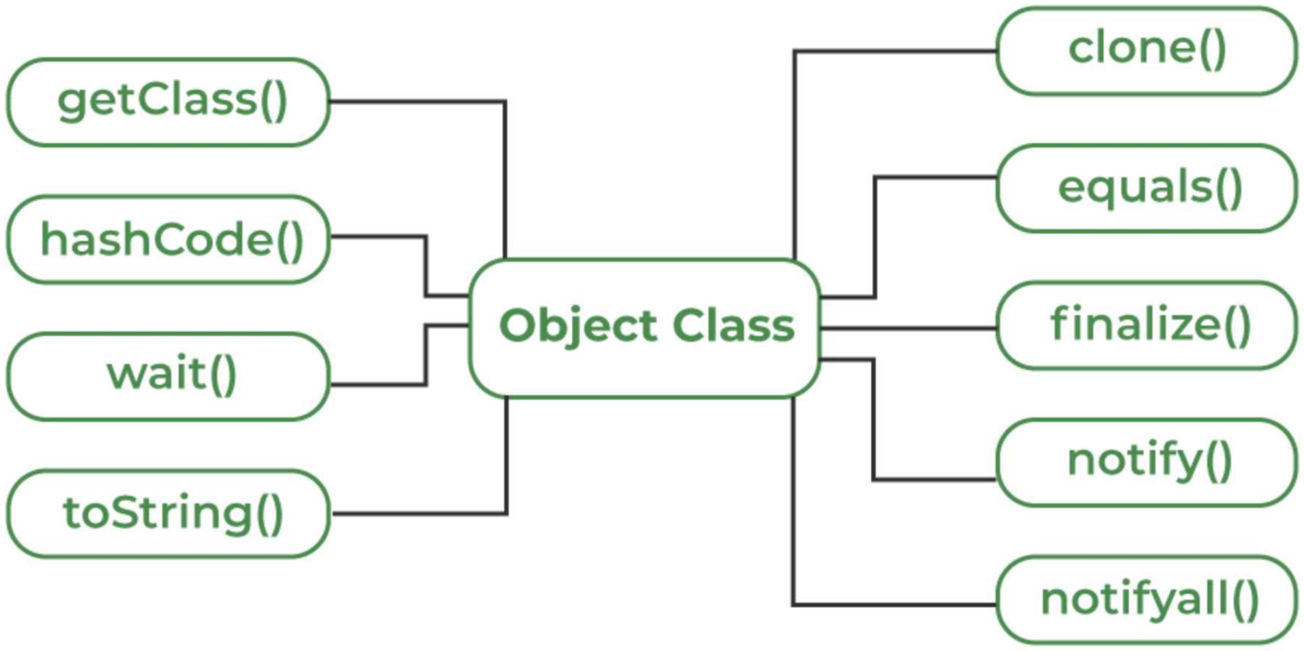
```
         Object
           ^
           |
         Animal
           ^
           |
         Mammal
```

Here, `Mammal` is a direct subclass of `Animal`, and `Animal` is a direct subclass of `Object`. Therefore, `Mammal` is also an indirect subclass of `Object`.

In summary, the concept of a superclass in Java allows for code reuse through inheritance. All classes in Java are implicitly or explicitly derived from the `Object` class, making it the root of the class hierarchy and the ultimate superclass for all other classes.

## Methods in Topmost Class Object

The `Object` class is a fundamental class provided by the Java standard library. It serves as the root of the class hierarchy and is the superclass of all other classes in Java. **Every class you create in Java implicitly or explicitly inherits from the `Object` class.** The `Object` class provides some essential methods that are available to all Java objects. Let's explore some scenarios and examples to help you understand the `Object` class better:



## Method 1: hashCode()

For every object, JVM generates a number which is a hashcode. It returns distinct integers for distinct objects (Really?! Let's explore how hash code generated). The `Object` class provides a default implementation of the `hashCode()` method that **returns an integer hash code based on the memory address of the object**. A common misconception about this method is that the hashCode() method returns the address of the object, which is not correct. It converts the internal address of the object to an integer by using an algorithm.

**hashCode() is a native method in Java** because in Java it is impossible to find the address of an object. so it uses native languages like C/C++ to find the address of the object.

The `hashCode()` is a method inherited from the `Object` class. It returns an integer value representing the hash code of an object.

### Custom Class

How about the custom class? How to implement hashCode to represent its objects?

`Student` class has 2 attributes, `id` and `name` to represent a student.

```
class Student {
    private String id;
    private String name;

    public Student(String id, String name) {
        this.id = id;
        this.name = name;
    }

```

```java
    // implicitly inherit the hashcode() method from Object.class
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("1001", "Alice");
        Student s2 = new Student("1001", "Alice");

        System.out.println(s1.hashCode()); // represents the object reference of s1
        System.out.println(s2.hashCode()); // represents the object reference of s2
        System.out.println(s1.hashCode() == s2.hashCode()); // false
    }
}
```

```java
// Source code - Object.class
@IntrinsicCandidate
public native int hashCode(); // implemented in JVM
```

In Java, `Class` is used to describe the world. **Assume id is the key to identifying a student in the world you are going to describe, we should use a way to describe their equality, even if they are really different objects in heap memory.**

In `Class`, we can use `equals()` and `hashCode()` methods to implement the above idea.

Let's rewrite the class in this way:

```java
class Student {
    private String id;
    private String name;

    public Student(String id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override // Overwrite the hashCode() in Object.class
    public int hashCode() {
        return Objects.hash(this.id);
    }
}

public class Main {
    public static void main(String[] args) {
        Student s1 = new Student("1001", "Alice");
        Student s2 = new Student("1001", "Alice");

        System.out.println(s1.hashCode()); // represents the object reference of s1
        System.out.println(s2.hashCode()); // represents the object reference of s2
        System.out.println(s1.hashCode() == s2.hashCode()); // false
    }
}
```

For other classes, such as `wrapper classes` and `String`, they have implemented `hashCode()` and `equals()` methods to describe itself.

For example, What is an identical String? What is an identical Integer? Obviously, the value of the String & Integer is the only key to identifying itself.

## String.class

```java
String str = "ABC";
System.out.println(str.hashCode()); // 64578
System.out.println(65 * (int) Math.pow(31, 2) + 66 * 31 + 67); // 64578

// A*31^2 + B*31 + C (as ASCII value of A = 65 and B = 66 and C = 67)
// = 65 * Math.pow(31,2) + 66 * 31 + 67
// = 64578
```

```java
// Source Code - StringLatin1.class
public static int hashCode(byte[] value) {
    int h = 0;
    for (byte v : value) {
        h = 31 * h + (v & 0xff);
    }
    return h;
}
```

## Integer.class

```java
Integer i1 = 200;
Integer i2 = 200;
```

```java
System.out.println(i1.hashCode() == i2.hashCode()); // true, they are 200
System.out.println(i1 == i2); // false, they are different objects
```

```java
// Source Code - Integer.class
public static int hashCode(int value) {
    return value;
}
```

Here we are not going through all wrapper class's hashCode() method. Check it out if you are interested.

**Note 1:** In Java, why is the hashCode() method provided for developers to check object equality, but not use the object reference directly?

**Note 2:** The equals() method is used to check if the objects are identical in the real world, but what is meaning of hash code? Who cares? Who uses this method?

We will explain all the above in the chapter Data Structure - Hashmap later. **At this stage, you just need to understand the difference between instances in heap memory and objects in the world. Besides, as best practice, you should overwrite both equals() and hashCode() in custom class, making sure they are consistent.**

## Method 2: equals()

The `equals()` method is another method inherited from the `Object` class. It is used to compare objects for equality. By default, the `equals()` method in the `Object` class checks if two object references point to the same memory location (i.e., if they are the same object). However, just like `toString()`, you can override this method to define your custom equality logic for your class.

```java
class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

}

public class Main {
    public static void main(String[] args) {
        Point p1 = new Point(2, 3);
        Point p2 = new Point(2, 3);
        // This will call the equals() method inherited from Object.class
        System.out.println(p1.equals(p2));  // false, they are different objects.
        // But, is this result fit your design?
        // Can it solve your problem?
        // Can this design describe two-dimensional coordinate geometry in our world?
    }
}
```

**Remember, equals() method is a way provided by Java, which allows us to describe the world more easily.**

## Point Example

```java
class Point {
    private int x;
    private int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    // Override the equals() method to provide custom equality logic
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (!(obj instanceof Point))
            return false;
        Point point = (Point) obj;
        return this.x == point.x
            && this.y == point.y;
        // You can write this approach:
        // return Objects.equals(this.x, this.y);
    }
}

public class Main {
    public static void main(String[] args) {
        Point p1 = new Point(2, 3);
        Point p2 = new Point(2, 3);
        // This will call the overridden equals() method
        System.out.println(p1.equals(p2));  // true
        // In two-dimensional coordinate geometry, (2, 3) is a point, which is unique.
        // As a user, I don't care the objects in heap,
        // I just want to mention the Point (coordinate) in geometry
    }
}

```

**Always overwrite hashCode() and equals() together, to let the object equality check become consistent. It is necessary to enhance readability and reduce the possibility to mislead the code reader. More importantly, hash-based data structure will invoke both equals() and hashCode() methods to perform data write & read operations.**

## Method 3: toString()

The `toString()` method is one of the methods inherited from the `Object` class. It is used to return a string representation of an object. By default, the `toString()` method in the `Object` class returns a string consisting of **the name of the class of which the object is an instance**, the **at-sign character @**, and the **unsigned hexadecimal representation of the hash code of the object**. However, you can override this method in your custom classes to provide a more meaningful string representation.

```
// Default behavior of toString() is to print class name, then
// @, then unsigned hexadecimal representation of the hash code
// of the object

public String toString() {
    return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

**Note 1: Whenever we try to print any Object reference, then internally the toString() method is called.**

That is why it prints object reference when we try to print out the variable of an array. The reason is, the default implementation of array is controlled by Java. It inherits the toString() method from Object Class.

```
Student s = new Student();

// Below two statements are equivalent
```

```java
System.out.println(s); // toString() is called internally, String representation of hashc code
System.out.println(s.toString());  // String representation of hashc code
```

***Note2:*** It is always recommended to override the **toString()** method to get our own String representation of Object. We often check the values of the object, but we don't care about the reference of the object.

```java
class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    // Override the toString() method to provide a custom string representation
    @Override
    public String toString() {
        return "Person: " + name + ", Age: " + age;
    }
}

public class Main {
    public static void main(String[] args) {
        Person person = new Person("Vincent Lau", 30);
        System.out.println(person); // This will call the overridden toString() method.
    }
}
```

## Method 4: getClass()

In Java, the `Object.getClass()` method is a built-in method provided by the `Object` class, which is the root class for all other classes in Java. It allows you to retrieve the runtime class of an object, which represents the actual class type of the object at runtime.

The `getClass()` method returns an object of type `Class<?>`, which represents the runtime class of the object. The `Class` class provides various methods to obtain information about the class, such as its name, superclass, implemented interfaces, constructors, methods, and more.

```java
public class Demo {
    public static void main(String[] args) {
        // Create an instance of the Person class
        Person person = new Person("Vincent", 30);

        // Use getClass() to get the runtime class of the object
        Class<?> childClass = person.getClass();

        // Print the class name
        System.out.println("Runtime class: " + childClass.getName());

        // Get the superclass of the runtime class
        Class<?> superClass = childClass.getSuperclass();
        System.out.println("Superclass: " + superClass.getName());
    }
}

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

Output:

```
Runtime class: Person
Superclass: java.lang.Object
```

In this example, we create an instance of the `Person` class and then use the `getClass()` method on that object to obtain the runtime class of the `person` object. We then print the name of the class, which is "Person", in this case. We also retrieve the superclass of the runtime class using the `getSuperclass()` method and print its name, which is "java.lang.Object", because `Person` is a direct subclass of `Object`.

The `getClass()` method is useful in scenarios where you need to perform **runtime type checks**, reflection, or when you want to obtain information about the object's class dynamically during the program's execution. However, it is important to use it judiciously, as excessive use of reflection **can lead to reduced performance and may complicate the code unnecessarily** in many cases.