# Object Class and other topics

## 9.7 `Object` Class

- All classes in Java inherit directly or indirectly from `Object`, so its 11 methods are inherited by all other classes.
- Figure 9.12 summarizes `Object`’s methods.
- Every array has an overridden `clone` method that copies the array.
  - If the array stores references to objects, the objects are not copied—a *shallow copy* is performed.

| Method | Description |
|---|---|
| `clone` | This `protected` method, which takes no arguments and returns an `Object` reference, makes a copy of the object on which it's called. The default implementation performs a so-called **shallow copy**—instance-variable values in one object are copied into another object of the same type. For reference types, only the references are copied. A typical overridden `clone` method's implementation would perform a **deep copy** that creates a new object for each reference-type instance variable. Implementing `clone` correctly is difficult. For this reason, its use is discouraged. Many industry experts suggest that object serialization should be used instead. We discuss object serialization in Chapter 17, Files, Streams and Object Serialization. |

**Fig. 9.12** | `Object` methods. (Part 1 of 3.)

| Method | Description |
|---|---|
| `equals` | This method compares two objects for equality and returns `true` if they're equal and `false` otherwise. The method takes any `Object` as an argument. When objects of a particular class must be compared for equality, the class should override method `equals` to compare the *contents* of the two objects. For the requirements of implementing this method, refer to the method's documentation at `download.oracle.com/javase/6/docs/api/java/lang/Object.html# equals(java.lang.Object)`. The default `equals` implementation uses operator == to determine whether two references *refer to the same object* in memory. Section 16.3.3 demonstrates class `String`'s `equals` method and differentiates between comparing `String` objects with == and with `equals`. |
| `finalize` | This `protected` method (introduced in Section 8.10) is called by the garbage collector to perform termination housekeeping on an object just before the garbage collector reclaims the object's memory. Recall that it's unclear whether, or when, method `finalize` will be called. For this reason, most programmers should avoid method `finalize`. |

**Fig. 9.12** | `Object` methods. (Part 2 of 3.)

| Method | Description |
|---|---|
| `getClass` | Every object in Java knows its own type at execution time. Method `getClass` (used in Sections 10.5, 14.5 and 24.3) returns an object of class `Class` (package `java.lang`) that contains information about the object's type, such as its class name (returned by `Class` method `getName`). |
| `hashCode` | Hashcodes are `int` values that are useful for high-speed storage and retrieval of information stored in a data structure that's known as a hashtable (discussed in Section 20.11). This method is also called as part of class `Object`'s default `toString` method implementation. |
| `wait`, `notify`, `notifyAll` | Methods `notify`, `notifyAll` and the three overloaded versions of `wait` are related to multithreading, which is discussed in Chapter 26. |
| `toString` | This method (introduced in Section 9.4.1) returns a `String` representation of an object. The default implementation of this method returns the package name and class name of the object's class followed by a hexadecimal representation of the value returned by the object's `hashCode` method. |

**Fig. 9.12** | `Object` methods. (Part 3 of 3.)

- **toString()** : toString() provides String representation of an Object and used to convert an object to String. The default toString() method for class Object 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.

- It is always recommended to override **toString()** method to get our own String representation of Object.

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

- Java Object **hashCode()** is a native method and returns the integer hash code value of the object. The general contract of hashCode() method is:
  - Multiple invocations of hashCode() should return the same integer value, unless the object property is modified that is being used in the equals() method.
  - An object hash code value can change in multiple executions of the same application.
  - If two objects are equal according to equals() method, then their hash code must be same.
  - If two objects are unequal according to equals() method, their hash code are not required to be different. Their hash code value may or may-not be equal.

- **Use of hashCode() method :** Returns a hash value that is used to search object in a collection. JVM(Java Virtual Machine) uses hashcode method while saving objects into hashing related data structures like HashSet, HashMap, Hashtable etc. The main advantage of saving objects based on hash code is that searching becomes easy.
  **Note :** Override of **hashCode()** method needs to be done such that for every object we generate a unique number. For example, for a Student class we can return roll no. of student from hashCode() method as it is unique.

- **equals(Object obj)** : Compares the given object to “this” object (the object on which the method is called). It gives a generic way to compare objects for equality.
- Use equals instead of == for reference types
- It is recommended to override **equals(Object obj)** method to get our own equality condition on Objects.
- **Note :** It is generally necessary to override the **hashCode()** method whenever this method is overridden, so as to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.

- **getClass()** : Returns the class object of “this” object and used to get actual runtime class of the object. It can also be used to get metadata of this class.
- As it is final we can’t override it.

## clone()

- An *assignment* copies the reference of an instance to a variable. A *clone* operation will clone the instance (and assign a reference to the clone).
- With *assignment*, you'll end up with *multiple* variables pointing to *one* object, with cloning you'll have *multiple* variables that hold references of *multiple* objects
- **Clone() returns the shallow copy of the object t1.**

- **Deep Copy vs Shallow Copy**

- **Shallow copy** is the method of copying an object and is followed by default in cloning. In this method, the fields of an old object X are copied to the new object Y.
- While copying the object type field the reference is copied to Y i.e object Y will point to the same location as pointed out by X.
- If the field value is a primitive type it copies the value of the primitive type.

- Therefore, any changes made in referenced objects in object X or Y will be reflected in other objects.
- *Shallow copies are cheap and simple to make.*

## Usage of clone() method – Deep Copy

- If we want to create a deep copy of object X and place it in a new object Y then a new copy of any referenced objects fields are created and these references are placed in object Y.
- This means any changes made in referenced object fields in object X or Y will be reflected only in that object and not in the other. In the below example, we create a deep copy of the object.

- A deep copy copies all fields and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

# Initialization Blocks

- You have already seen two ways to initialize a data field:
  - By setting a value in a constructor
  - By assigning a value in the declaration
- There is a third mechanism in Java, called an initialization block. Class declarations can contain arbitrary blocks of code. These blocks are executed whenever an object of that class is constructed. For example:

```
class Employee
{
   private static int nextId;
   private int id;
   private String name;
   private double salary;
   // object initialization block
   {
     id = nextId;
     nextId++;
   }
```

```
public Employee(String n, double s)
{
name = n;
salary = s;
}
public Employee()
{
name = "";
salary = 0;
}
. . .
}
```

- In this example, the id field is initialized in the object initialization block, no matter which constructor is used to construct an object. The initialization block runs first, and then the body of the constructor is executed.
- This mechanism is never necessary and is not common. It is usually more straightforward to place the initialization code inside a constructor.

- With so many ways of initializing data fields, it can be quite confusing to give all possible pathways for the construction process. Here is what happens in detail when a constructor is called:

1. If the first line of the constructor calls a second constructor, then the second constructor executes with the provided arguments.

2. Otherwise,

    a. All data fields are initialized to their default values (0, false, or null).

    b. All field initializers and initialization blocks are executed, in the order in which they occur in the class declaration.

3. The body of the constructor is executed.

# Static initialization

- To initialize a static field, either supply an initial value or use a static initialization block. You have already seen the first mechanism:
    - private static int nextId = 1;
- If the static fields of your class require complex initialization code, use a static initialization block.
- Place the code inside a block and tag it with the keyword static. Here is an example. We want the employee ID numbers to start at a random integer less than 10,000.

```
// static initialization block
static
{
     var generator = new Random();
     nextId = generator.nextInt(10000);
}
```

- Static initialization occurs when the class is first loaded. Like instance fields, static fields are `0`, `false`, or `null` unless you explicitly set them to another value. All static field initializers and static initialization blocks are executed in the order in which they occur in the class declaration.

- Amazingly enough, up to JDK 6, it was possible to write a “Hello, World” program in Java without ever writing a main method.

```
public class Hello
{
 static
 {
        System.out.println("Hello, World");
 }
}
```

- When you invoked the class with java Hello, the class was loaded, the static initialization block printed “Hello, World”, and only then was a message displayed that main is not defined. Since Java 7, the java program first checks that there is a main method.

## Object Destruction and the **finalize** Method

- Some object-oriented programming languages, notably C++, have explicit destructor methods for any cleanup code that may be needed when an object is no longer used. The most common activity in a destructor is reclaiming the memory set aside for objects.
- Since Java does automatic garbage collection, manual memory reclamation is not needed, so Java does not support destructors.
- some objects utilize a resource other than memory, such as a file or a handle to another object that uses system resources. In this case, it is important that the resource be reclaimed and recycled when it is no longer needed.
- If a resource needs to be closed as soon as you have finished using it, supply a `close` method that does the necessary cleanup. You can call the `close` method when you are done with the object.

## Example

- The **close()** method ofjava.util.Scanner class closes the scanner which has been opened. If the scanner is already closed then on calling this method, it will have no effect.

```
Scanner scanner = new Scanner(s);
String str = scanner.nextLine();
scanner.close();
```

**Caution**

Do not use the `finalize` method for cleanup. That method was intended to be called before the garbage collector sweeps away an object. However, you simply cannot know when this method will be called, and it is now deprecated.