Object as a Superclass

Methods From the Object Class

The Object class, in the `java.lang` package, sits at the top of the class hierarchy tree. Every class is a descendant, direct or indirect, of the `Object` class. Every class you use or write inherits the instance methods of `Object`. You need not use any of these methods, but, if you choose to do so, you may need to override them with code that is specific to your class. The methods inherited from `Object` that are discussed in this section are:

- `protected Object clone() throws CloneNotSupportedException`: Creates and returns a copy of this object.
- `public boolean equals(Object obj)`: Indicates whether some other object is "equal to" this one.
- `protected void finalize() throws Throwable`: Called by the garbage collector on an object when garbage collection determines that there are no more references to the object. As of Java 18, this method has been deprecated for removal.
- `public final Class getClass()`: Returns the runtime class of an object.
- `public int hashCode()`: Returns a hash code value for the object.
- `public String toString()`: Returns a string representation of the object.

Note that, as of Java SE 9 the `finalize()` method had been deprecated, and deprecated for removal as of Java SE 18. Overriding this method is strongly discouraged. At some point it will not be called anymore. See the last section of this page for more information.

The `notify()`, `notifyAll()`, and `wait()` methods of `Object` all play a part in synchronizing the activities of independently running threads in a program, which is discussed in a later section and will not be covered here. There are five of these methods:

- `public final void notify()`
- `public final void notifyAll()`
- `public final void wait()`
- `public final void wait(long timeout)`
- `public final void wait(long timeout, int nanos)`

> *Note: There are some subtle aspects to a number of these methods, especially the clone method.*

The toString() Method

You should always consider overriding the `toString()` method in your classes.

The Object's `toString()` method returns a `String` representation of the object, which is very useful for debugging. The `String` representation for an object depends entirely on the object, which is why you need to override `toString()` in your classes.

You can use `toString()` along with `System.out.println()` to display a text representation of an object, such as an instance of `Book`:

```
System.out.println(firstBook.toString());
```

which would, for a properly overridden `toString()` method, print something useful, like this:

```
ISBN: 0201914670; The Swing Tutorial; A Guide to Constructing GUIs, 2nd Edition
```

The equals() Method

The `equals()` method compares two objects for equality and returns true if they are equal. The `equals()` method provided in the `Object` class uses the identity operator (`==`) to determine whether two objects are equal. For primitive data types, this gives the correct result. For objects, however, it does not. The `equals()` method provided by `Object` tests whether the object references are equal—that is, if the objects compared are the exact same object.

To test whether two objects are equal in the sense of equivalency (containing the same information), you must override the `equals()` method. Here is an example of a `Book` class that overrides `equals()`:

```java
public class Book {
    String ISBN;

    public String getISBN() {
        return ISBN;
    }

    public boolean equals(Object obj) {
        if (obj instanceof Book)
            return ISBN.equals((Book)obj.getISBN());
        else
            return false;
    }
}
```

Consider this code that tests two instances of the `Book` class for equality:

```java
// Swing Tutorial, 2nd edition
Book firstBook  = new Book("0201914670");
Book secondBook = new Book("0201914670");
if (firstBook.equals(secondBook)) {
    System.out.println("objects are equal");
} else {
    System.out.println("objects are not equal");
}
```

This program displays objects are equal even though `firstBook` and `secondBook` reference two distinct objects. They are considered equal because the objects compared contain the same ISBN number.

You should always override the `equals()` method if the identity operator is not appropriate for your class.

> *Note: If you override `equals()`, you must override `hashCode()` as well.*

The hashCode() Method

The value returned by `hashCode()` is the object's hash code, which is an integer value generated by a hashing algorithm.

By definition, if two objects are equal, their hash code must also be equal. If you override the `equals()` method, you change the way two objects are equated and `Object`'s implementation of `hashCode()`is no longer valid. Therefore, if you override the `equals()` method, you must also override the `hashCode()` method as well.

The getClass() Method

You cannot override `getClass()`.

The `getClass()` method returns a `Class` object, which has methods you can use to get information about the class, such as its name (`getSimpleName()`), its superclass (`getSuperclass()`), and the interfaces it implements (`getInterfaces()`). For example, the following method gets and displays the class name of an object:

```
void printClassName(Object obj) {
    System.out.println("The object's" + " class is " +
        obj.getClass().getSimpleName());
}
```

The `Class` class, in the `java.lang` package, has a large number of methods (more than 50). For example, you can test to see if the class is an annotation (`isAnnotation()`), an interface (`isInterface()`), or an enumeration (`isEnum()`). You can see what the object's fields are (`getFields()`) or what its methods are (`getMethods()`), and so on.

The clone() Method

If a class, or one of its superclasses, implements the `Cloneable` interface, you can use the `clone()` method to create a copy from an existing object. To create a clone, you write:

```
aCloneableObject.clone();
```

`Object`'s implementation of this method checks to see whether the object on which `clone()` was invoked implements the `Cloneable` interface. If the object does not, the method throws a

CloneNotSupportedException exception. Exception handling will be covered in the section Exception. For the moment, you need to know that clone() must be declared as

```
protected Object clone() throws CloneNotSupportedException
```

or

```
public Object clone() throws CloneNotSupportedException
```

if you are going to write a clone() method to override the one in Object.

If the object on which clone() was invoked does implement the Cloneable interface, Object's implementation of the clone() method creates an object of the same class as the original object and initializes the new object's member variables to have the same values as the original object's corresponding member variables.

The simplest way to make your class cloneable is to add `implements Cloneable` to your class's declaration. then your objects can invoke the clone() method.

For some classes, the default behavior of Object's clone() method works just fine. If, however, an object contains a reference to an external object, say `ObjExternal`, you may need to override clone() to get correct behavior. Otherwise, a change in `ObjExternal` made by one object will be visible in its clone also. This means that the original object and its clone are not independent—to decouple them, you must override clone() so that it clones the object and `ObjExternal`. Then the original object references `ObjExternal` and the clone references a clone of `ObjExternal`, so that the object and its clone are truly independent.

The finalize() Method

The Object class provides a callback method, finalize(), that may be invoked on an object when it becomes garbage. Object's implementation of finalize() does nothing. Overriding finalize() was made to do some clean-up, such as freeing resources.

The finalize() method may be called automatically by the system, but when it is called, or even if it is called, is uncertain. Therefore, you should not rely on this method to do your cleanup for you anymore. For example, if you do not close file descriptors in your code after performing I/O and you expect finalize() to close them for you, you may run out of file descriptors.

As of Java SE 9 the `finalize()` method had been deprecated, and as of Java SE 18, deprecated for removal. At some point, it will not be called anymore. Overriding this method is now strongly discouraged. If you need to clean up some resources, you may do so by implementing the `AutoCloseable` interface. This point is covered in detail in the Java I/O section.

Last update: *September 14, 2021*

Polymorphism ← **Object as a Superclass** → Abstract Methods and Classes

Home > Tutorials > Inheritance > Object as a Superclass