The Object clone() Method

Introduction

In this article from my free Java 8 course, I will be discussing the Java Object `clone()` method. The `clone()` method is defined in class Object which is the superclass of all classes. The method can be used on any object, but generally, *it is not advisable to use this method.*

The clone() Method

The `clone()` method, as the name implies, creates an identical copy of an object. Depending on the type of implementation, this copy can either be a shallow or a deep copy. In any case, the objects will be equal, but not identical (For more details see my article about Identity and Equality in Java).

Using *clone()* on an Object

Let's take a look at how the `clone()` method works. In Example 1 we create an object of class `Porsche` named `marcusPorsche`, giving me a very nice new car. However, I'm a nice guy and I'd like to give away another Porsche; so I'll clone the `marcusPorsche` object and assign the new object to the reference variable `peterPorsche`. If your name is Peter, congratulations! You just got a new Porsche!

```
package com.marcusbiel.java8course;

public class PorscheTest {

     @Test
     public void shouldClonePorsche(){
          Porsche marcusPorsche = new Porsche();
          Porsche peterPorsche = porsche.clone();
          assertNotSame(porsche, peterPorsche);
     }
}
```

Example 1

However, something is still wrong. The `clone()` method is red. The problem is that the `clone()` method from class is `Object` protected. Every object can call `protected` methods inherited from the Object class *on itself*. However, it can never call such a method on *other objects.* While `clone()` is accessible to *both* our `Porsche` class and our `PorscheTest` class, the `PorscheTest` can never call the inherited `clone()` method of a `Porsche` object.

To fix this problem, we have to override the `clone()` method in the Porsche class and increase its visibility. First, we change the access modifier of the `clone()` method to `public` and change the method return type from `Object` to `Porsche`, which makes our code clearer since we don't have to cast cloned objects into Porsches. To implement the clone method, we call `super.clone()`. The super keyword means that we are calling a method from a superclass, which in this case is the Object class.

Note also that the clone method of the Object class is declaring a checked `CloneNotSupportedException`, so we have to decide whether we want to declare it or handle it. Declaring a `CloneNotSupportedException` while implementing (supporting) the `clone()` method is contradictory. Therefore, you should omit it. All possible error situations are serious errors, so the best you can possibly do is to throw an `Error` instead.

```
package com.marcusbiel.java8course;

public class Porsche implements Car {

  @Override
  public Porsche clone(){
        try{
           return(Porsche)super.clone();
        } catch(CloneNotSupportedException e){
            throw new AssertionError(); /* can never happen */
        }
  }
}
```

Example 2

However, when we run the above code, we encounter another issue: a `CloneNotSupportedException`. To correct that, we have to implement the `Cloneable` interface. The `Cloneable` interface does not contain any methods, it is a Marker Interface - an empty interface used to mark some property of the class that implements it.

```
public class Porsche implements Car, Cloneable{
```

Example 3

Now when we run the test in Example 1, it passes successfully.

Modifying an Object after clone()

At this point, we're going to add a test to check the content of the new object. We want to verify that our owner has been correctly identified in each object. The `asString()` method will be used to return a `String` representation of our `Porsche` object. The expected result when we are finished is for the object to be "`Porsche of Peter`". First, I'm going to create the `asString()` method in our `Porsche` class and an owner attribute.

```
String ownerName;

[...]

public String asString(){
     return "Porsche of" + ownerName;
}
```

Example 4

In the code below, the `assertEquals()` function is used to compare the output of the `asString()` function. When we run the test, it will fail, as the owner of both Porsches is still "`Marcus`".

```
@Test
public void shouldClonePorsche(){
     Porsche marcusPorsche = new Porsche("Marcus");
     Porsche peterPorsche = porsche.clone();
     assertNotSame(porsche, peterPorsche);

     assertEquals("Porsche of Peter", peterPorsche.asString());
}
```

Example 5

To fix this test, we need to create a method for transferring ownership of the `car`. I'm going to call this method `sellTo()`.

```
public void sellTo(String newOwnerName){
     ownerName = newOwnerName;
}
```

Example 6

Now I will call the sellTo() method on the cloned Porsche object, to transfer ownership of the cloned Porsche to Peter. As a final proof that cloning the Porsche object created a **fully independent** second Porsche object, I will test that transferring the ownership to Peter on our cloned Porsche object did not influence the original Porsche object. In other words, I will test whether the original Porsche object still belongs to Marcus or not.

```
@Test
public void shouldClonePorsche(){
     Porsche marcusPorsche = new Porsche("Marcus");
     Porsche peterPorsche = porsche.clone();
     assertNotSame(porsche, peterPorsche);

     peterPorsche.sellTo("Peter");
     assertEquals("Porsche of Marcus", porsche.asString());
     assertEquals("Porsche of Peter", peterPorsche.asString());
}
```

Example 7

This time, when we run the test, it will pass. This proves that we have created two independent objects that can be changed independently, so our cloning method works as expected.

Using clone() on an Array

As I've said before, it is generally not advisable that the clone method be used. However, one case where I *do* recommend its use is for cloning arrays, as shown in example 8. Here, we create an array of type `String`. We call the `array.clone()` method, using our array, and assign the new cloned array to a reference variable called `copiedArray`. To prove that this not just a reference to the same object, but a new, independent object, we call the `assertNotSame()` with our original `array` and our newly created `copiedArray`. Both arrays have the same content, as you can see when we print out the `copiedArray` in a for-each loop.

```
@Test
public void shouldCloneStringArray() {
     String[] array = {“one”,“two”,“three”};
     String[] copiedArray = array.clone();
     assertNotSame(array, copiedArray);
     for(String str: copiedArray){
          System.out.println(str);
     }
}
```

Example 8

The `clone()` method copies every string object in the array into a new array object that contains completely new string objects. If you ran the code above, the `clone()` method would work as expected and the test would be green. In this case, the `clone()` method is the prefered technique for copying an array. `Clone()` works great with arrays of primitive values and “immutables”, but it doesn’t work as well for arrays of objects. As a side note, in the Java Specialists’ Newsletter Issue 124, by Heinz Kabutz, a test was performed that showed that the `clone()` method is a bit slower for copying very small arrays, but for large arrays, where performance matters most, it’s actually faster than any other method of copying arrays.

Alternatives to Clone(): The Copy Constructor

There are two recommended alternatives to using the `clone()` method that deal with the shortcomings of `clone()`. The first method is using a **copy constructo**r that accepts one parameter - the object to clone. A copy constructor is really nothing very special. As you can see in Example 8, it is just a simple constructor that expects an argument of the class it belongs to.
The constructor then copies (clones) all sub-objects. If the new object just references the old sub-objects, we call it a shallow copy. If the new object references truly copied objects, we call it a deep copy. You can learn about this topic by reading my article Shallow vs Deep Copy.

```
package com.marcusbiel.java8course;

public class BMW implements Car, Cloneable {

     private Name ownersName;
     private Color color;

     public BMW(BMW bmw) {
          this.ownersName = new Name(bmw.ownersName);
          this.color = new Color(bmw.color);
     }
}
```

Example 9

Alternatives to Clone(): The static Factory Method

The other alternative is a static factory method. As the name implies, a static factory method is a static method, used to create an instance of the class. It can also be used to create a clone. There are a few common names for static factory methods; I'm going to use `newInstance()`. I've also created the same method for Name and Color, in order to recursively perform the operation on all sub-objects.

```
package com.marcusbiel.java8course;

public class BMW implements Car, Cloneable {

     private Name ownersName;
     private Color color;
     public static BMW newInstance(BMW bmw){
          return new BMW(Name.newInstance(bmw.ownersName),
                    Color.newInstance(bmw.color));
     }
     public BMW(Name ownersName, Color color){
          this.ownersName = ownersName;
          this.color = color;
     }
}
```

Example 10

Both of these alternatives are always better than the `clone()` method and produce the same result: a clone of the original object. The static factory method might be a bit more powerful than the copy constructor in certain cases, but generally both lead to the same result.

Immutables

The last thing I'd like to do in this article is take a look at the `Name` class:

```
package com.marcusbiel.java8course.car;

public class Name implements Cloneable{
     private String firstName;
     private String lastName;

     public static Name newInstance(Name name){
          return new Name(name.firstName, name.lastName);
     }
}
```

Example 11

Notice however, that the `String` members are passed by reference, without creating a new object. The reason for that is that `String` is an Immutable object. An Immutable object will not be changed by the reference in the new object, and therefore can be safely used without cloning. This is an extremely important concept that applies when you are trying to create a Deep Copy. If you are interested, you can read more about creating Immutables in my article about Immutables which is coming soon!

Thanks for reading!