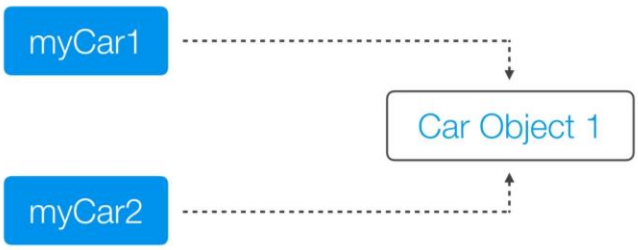
Shallow Copy vs. Deep Copy

Introduction

In this article from my free Java 8 Course, I will be discussing the difference between a Deep and a Shallow Copy.

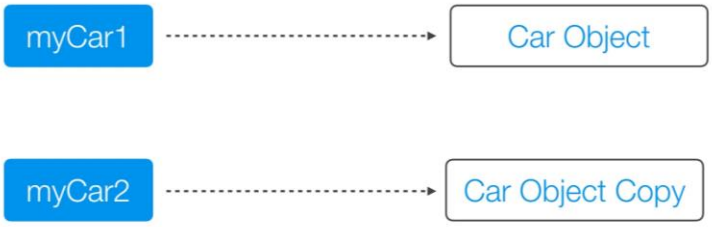
What is a Copy?

To begin, I'd like to highlight what a copy in Java is. First, let's differentiate between a reference copy and an object copy. A **reference copy**, as the name implies, creates a copy of a reference variable pointing to an object. If we have a Car object, with a `myCar` variable pointing to it and we make a reference copy, we will now have two `myCar` variables, but still one object.



Example 1

An **object copy** creates a copy of the object itself. So if we again copied our `car` object, we would create a copy of the object itself, as well as a second reference variable referencing that copied object.



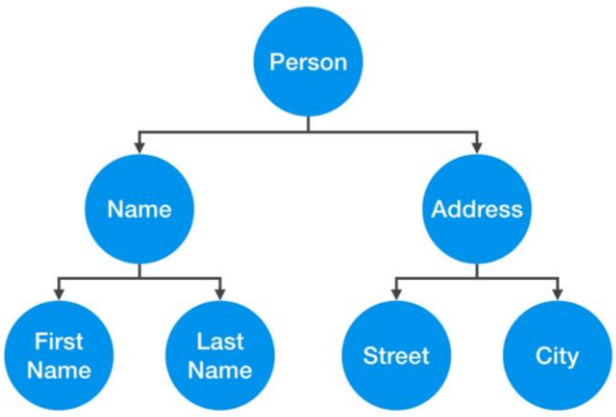
Example 2

What is an Object?

Both a Deep Copy and a Shallow Copy are types of object copies, but what really is an object? Often, when we talk about an object, we speak of it as a single unit that can't be broken down further, like a humble coffee bean. However, that's oversimplified.

Example 3

Say we have a `Person` object. Our `Person` object is in fact composed of other objects, as you can see in Example 4. Our `Person` contains a `Name` object and an `Address` object. The `Name` in turn, contains a `FirstName` and a `LastName` object; the `Address` object is composed of a `Street` object and a `City` object. So when I talk about `Person` in this article, I'm actually talking about this *entire network of objects*.



Example 4

So why would we want to copy this `Person` object? An object copy, usually called a clone, is created if we want to modify or move an object, while still preserving the original object. There are many different ways to copy an object that you can learn about in another article. In this article we'll specifically be using a copy constructor to create our copies.

Shallow Copy

First let's talk about the shallow copy. A shallow copy of an object copies the 'main' object, but doesn't copy the inner objects. The 'inner objects' are shared between the original object and its copy. For example, in our `Person` object, we would create a second `Person`, but both objects would share the same `Name` and `Address` objects.

Let's look at a coding example. In Example 5, we have our class `Person`, which contains a `Name` and `Address` object. The copy constructor takes the `originalPerson` object and copies its reference variables.

```
public class Person {
     private Name name;
     private Address address;

     public Person(Person originalPerson) {
          this.name = originalPerson.name;
          this.address = originalPerson.address;
     }
[…]
}
```

Example 5

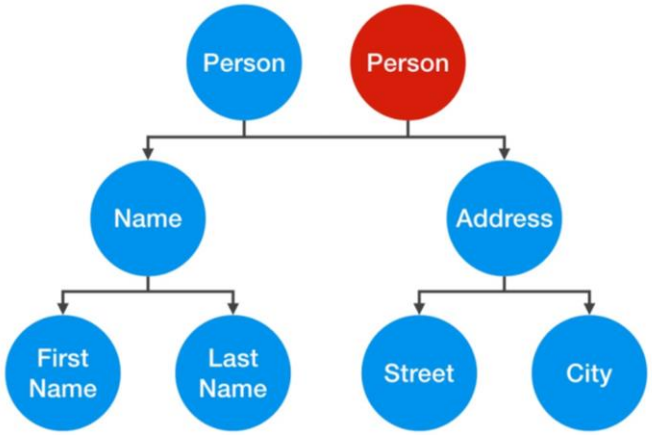
The problem with the shallow copy is that the two objects are not independent. If you modify the `Name` object of one `Person`, the change will be reflected in the other `Person` object.

Let's apply this to an example. Say we have a `Person` object with a reference variable `mother`; then, we make a copy of `mother`, creating a second `Person` object, `son`. If later on in the code, the `son` tries to `moveOut()` by modifying his `Address` object, the `mother` moves with him!

```
Person mother = new Person(new Name(…), new Address(…));
[…]
Person son  = new Person(mother);
[…]
son.moveOut(new Street(…), new City(…));
```

Example 6

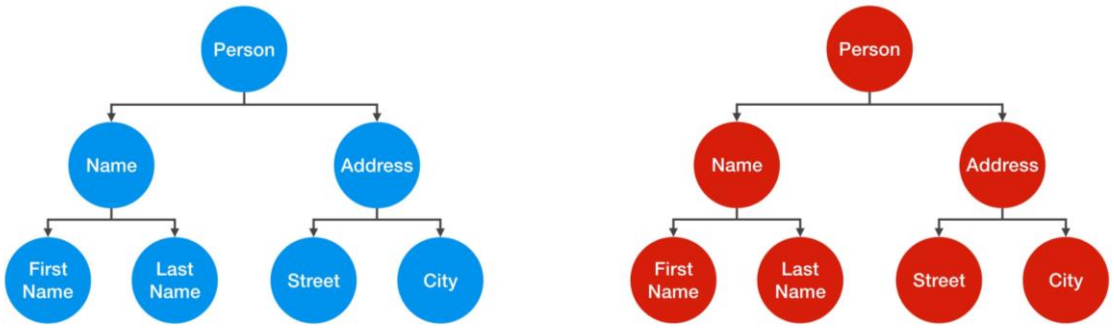
This occurs because our `mother` and `son` objects share the same `Address` object, as you can see illustrated in Example 7. When we change the `Address` in one object, it changes in both!



Example 7

Deep Copy

Unlike the shallow copy, a deep copy is a **fully independent copy of an object**. If we copied our `Person` object, we would copy the entire object structure.



Example 8

A change in the `Address` object of one `Person` wouldn't be reflected in the other object as you can see by the diagram in Example 8. If we take a look at the code in example 9, you can see that we're not only using a copy constructor on our `Person` object, but we are also utilizing copy constructors on the inner objects as well.

```
public class Person {
     private Name name;
     private Address address;

     public Person(Person otherPerson) {
          this.name    =  new Name(otherPerson.name);
          this.address =  new Address(otherPerson.address);
     }
[…]
}
```

Example 9

Using this deep copy, we can retry the mother-son example from Example 6. Now the `son` is able to successfully move out!

However, that's not the end of the story. To create a true deep copy, we need to keep copying all of the `Person` object's nested elements, until there are only primitive types and "Immutables" left. Let's look at the Street class to better illustrate this:

```
public class Street {
     private String name;
     private int number;

     public Street(Street otherStreet){
          this.name = otherStreet.name;
          this.number = otherStreet.number;
     }
[…]
}
```

Example 10

The `Street` object is composed of two instance variables - `String name` and `int number`. `int number` is a primitive value and not an object. It's just a simple value that can't be shared, so by creating a second instance variable, we are automatically creating an independent copy. `String` is an Immutable. In short, an Immutable is an Object, that, once created, can never be changed again. Therefore, you can share it without having to create a deep copy of it.

Conclusion

To conclude, I'd like to talk about some coding techniques we used in our mother-son example. Just because a deep copy will let you change the internal details of an object, such as the `Address` object, it doesn't mean that you should. Doing so **would decrease code quality**, as it would make the `Person` class more fragile to changes - whenever the `Address` class is changed, you will have to (potentially) apply changes to the `Person` class also. For example, if the `Address` class no longer contains a `Street` object, we'd have to change the `moveOut()` method in the `Person` class on top of the changes we already made to the `Address` class.

In Example 6 of this article I only chose to use a new `Street` and `City` object to better illustrate the difference between a shallow and a deep copy. Instead, I would recommend that you assign a new `Address` object instead, effectively converting to a ***hybrid*** of a shallow and a deep copy, as you can see in Example 10:

```
Person mother = new Person(new Name(…), new Address(…));
[…]
Person son  = new Person(mother);
[…]
son.moveOut(new Address(...));
```

Example 10

In object-oriented terms, this violates **encapsulation**, and therefore should be avoided. Encapsulation is **one of the most important aspects of Object Oriented programming**. In this case, I had violated encapsulation by accessing the internal details of the `Address` object in our `Person` class. This harms our code because we have now entangled the `Person` class in the `Address` class and if we make changes to the `Address` class down the line, it could harm the `Person` class as I explained above. While you obviously need to interconnect your various classes to have a coding project, whenever you connect two classes, you need to analyze the costs and benefits.

Thanks for reading!