Immutables

Introduction

In this article from my free Java 8 course, I will be discussing immutables in Java. The concept of immutability has always been important in all programming languages, including Java. With the release of Java 8 however, immutables have become even more important. This version introduced functional programming, as well as the new java.time api. Both rely heavily on immutables.

What is an Immutable?

An immutable class is a class whose instances cannot be modified. Information stored in an immutable object is provided when the object is created, and after that it is unchangeable and read-only forever. As we can't modify immutable objects, we need to work around this. For instance, if we had a spaceship class, and we wanted to change its location, we'd have to return a new object with modified information.

```
public Spaceship exploreGalaxy() {
     return new Spaceship(name, Destination.OUTER_SPACE);
}
```

Example 1

Advantages of Immutables

At first glance, you'd think that immutables were useless, however, they provide many advantages.

Firstly, immutable classes greatly reduce the effort needed to implement a stable and fault tolerant system. The property of immutables that prevents them from being changed is extremely beneficial when creating this kind of system.

Imagine that we have a `Bank` class that we are making for a very big bank. After the financial crisis, the bank is afraid of allowing their users to have a negative balance. So they institute a new rule and add a validation method to throw an `IllegalArgumentException` whenever a function call results in a negative balance. This type of rule is called an invariant.

```
public class BankAccount{
     [...]
     private void validate(long balance) {
          if (balance < 0) {
               throw new IllegalArgumentException("balance must not be
                                                   negative:"+ balance);
          }
     }
}
```

Example 2

In a typical class, this `validate()` method would be called anytime a user's balance is changed. If the user makes a withdrawl, pays their debt, or transfers money from their account we would have to call the validate method. However, with an immutable class we only have to call the validate method once, in the class constructor.

```
public BankAccount(long balance) {
     validate(balance);
     this.balance = balance;
}
```

Example 3

Since an immutable object will never change, this condition holds true for the entire lifetime of the object. Further validation will not be needed. Whenever a method that modifies the balance is called, a new object is returned, calling the constructor again and revalidating the object. This is extremely useful as it allows us to centralize all our invariants and guarantee that objects are consistent for their entire lifetime.

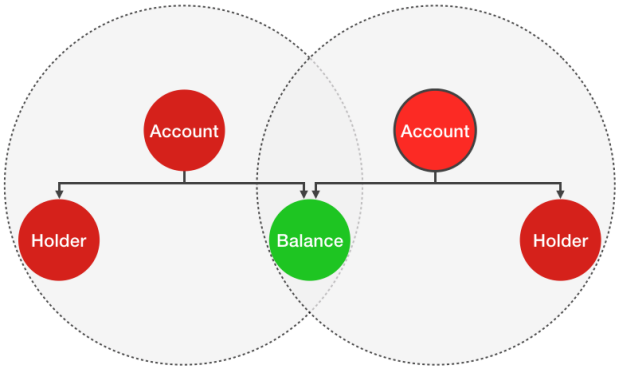
Similarly, immutables can be used to support a fault tolerant system. Imagine that you try to withdraw money from a bank, but between the time your money is withdrawn from your account and the money is released from the ATM, there is some error. In a normal class, your money would be gone forever. The account object was changed, too late. But in an immutable class, you can throw an error, preventing your account from losing money before you physically receive it.

```
public ImmutableAccount withdraw(long amount) {
     long newBalance = newBalance(amount);
     return new ImmutableAccount(newBalance);
}
private long newBalance(long amount) {
     // exception during balance calculation
}
```

Example 4

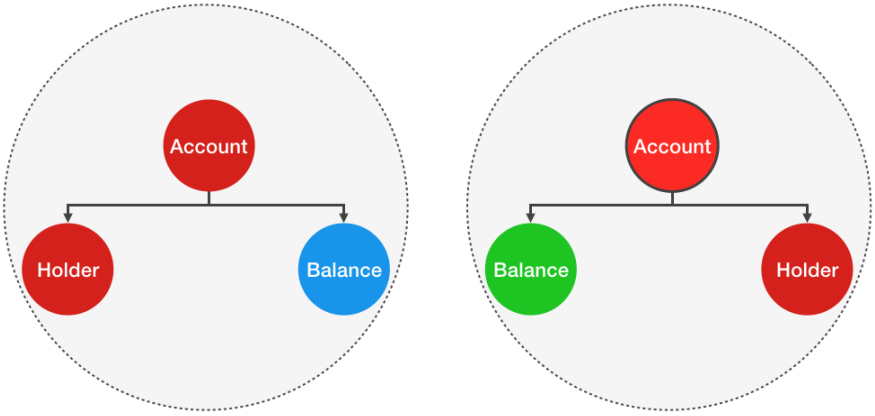
An immutable object can never get into an inconsistent state, even in the case of an exception. This stabilizes our system and removes the threat of an unforeseen error destabilizing an entire system. This stability comes at no cost, apart from the cost of the initial validation.

A second advantage of immutables is that they can be shared freely between objects. Let's say we create a copy of our account object. When we copy the object, we leave both objects sharing the same balance object.



Example 5

However, since the objects are both immutable, if we change the balance in one of the objects it doesn't affect the other object. The other object creates a new immutable instance of the class and the two objects are unaffiliated.



Example 6

For the same reason, an immutable does not need a copy constructor when copying an object. Immutables can even be shared freely when using a lock free algorithm in a multithreaded environment, where multiple actions happen in parallel.

Finally, immutable objects are also a perfect option to use as Map keys and Set elements, since Map keys and Set elements must never change.

Disadvantages of Immutables

As you've probably realized, the rigidness of immutables can be a huge asset, but it can also be a disadvantage. The biggest weakness of immutables is their potential to cause performance problems. Every time you have a new state of the class, you need to create a new object. Because of this, you usually need to create a lot more immutable objects than you would need to create mutable objects. Logically, the more objects you create, the more system resources you use.

This might be a problem, or it might not, it depends on a variety of factors. What are you trying to do? What kind of hardware is your program running? Are you building a desktop or a web application? How big is your program? The combination of these factors determines whether or not making your class immutable will cause performance issues. Typically, you should try to utilize immutables as much as possible. Start by making every class immutable and facilitate their immutability by creating small classes with simple methods. Simplicity and clean code are key. If you have clean code, that facilitates immutability; and if you have immutables, your code is cleaner. Once you have your program, test it. See how it is performing and if performance is not satisfactory, gradually relax the immutability rules.

How to Make an Immutable

Now that I've shown you why immutables are valuable and when you should use them, I'm going to show you how to make an immutable class. Let's go through turning our mutable class `Spaceship` into an immutable class:

```
public class Spaceship {
     public String name;
     public Destination destination;
     public Spaceship(String name) {
          this.name = name;
          this.destination = Destination.NONE;
     }
     public Spaceship(String name, Destination destination) {
          this.name = name;
          this.destination = destination;
     }
     public Destination currentDestination() {
          return destination;
     }
     public Spaceship exploreGalaxy() {
          destination = Destination.OUTER_SPACE;
     }

 […]
}
```

Example 7

To make a mutable class into an immutable class you should follow four steps:

1. Make all fields private and final
2. Don't provide any methods that modify the object's state
3. Ensure that the class can't be extended
4. Ensure exclusive access to any mutable fields

Make all fields private and final

The first step in making a mutable class immutable is changing all of its fields to be `private` and `final`. We make variables `private` so that they cannot be accessed from outside the class. If they were accessible from outside the class, they could be changed. We also make fields `final` to clearly convey that we never want them to be reassigned within the class. Should someone try to reassign a reference variable, a compiler error will occur.

```
private final String name;
private final Destination destination;
```

Example 8

Don't provide any methods that modify the object's state

The next step is to project our object's state from being modified. As we've talked about before, immutables, by definition, cannot have their object's state modified. Whenever you have a method that would modify an object's state, you instead have to return a new object.

```
public ImmutableSpaceship exploreGalaxy() {
     return new ImmutableSpaceship(name, Destination.OUTER_SPACE);
}
```
Example 9

Any fields that we aren't changing, such as `name`, can be directly copied from our current object. This is because, as I explained before, immutable objects can share fields freely. Fields that we do change need to be initialized as new objects.

Ensure that the class can't be extended

To prevent our class from being changed, we also need to protect our class from being extended. If a class can be extended we could override methods in the class. An overridden method could modify our object, which violates the rules of immutability. Let's look at an example in code:

```
public class EvilSpaceship extends Spaceship {
     [...]
     @Override
     public EvilSpaceship exploreGalaxy() {
          this.destination = Destination.OUTER_SPACE;
          return this;
     }
}
```
Example 10

To stop this `EvilSpaceship` from destroying our sacred immutability, make the class `final`.

```
public final class Spaceship
```
Example 11

Ensure exclusive access to mutable fields

The final step in ensuring immutability is to protect our access to mutable fields. Remember, immutable fields can be shared freely, so it doesn't matter if we have exclusive access. Anyone who holds access to a mutable field can alter it, thereby modifying our immutable object. To prevent anyone from directly accessing a mutable field, we should never obtain or return a direct reference to a `Destination` object. Instead we have to create a deep copy of our mutable object and work with that instead. As long as the mutable object is never directly shared, a change inside an external object will not have any effect on our immutable object. To achieve exclusive access we have to check all public methods and constructors for any incoming or outgoing `Destination` references.

The public constructor does not receive any `Destination` reference. The `Destination` object it creates is safe, as it cannot be accessed from outside. So the public constructor is good as it is.

In our `currentDestination()` method, however, we return our `Destination` object, which is a problem. Instead of returning the real reference, create a deep copy of our `Destination` object and return a reference to the copy.

```
public Destination currentDestination() {
     return new Destination(destination);
}
```

Example 12

The final public method we still have is the `newDestination()` method. It receives a Destination reference and directly forwards it to our constructor. This means that it is referencing the same object as whatever called this method. To guard against this, we can either make a deep copy within this method, or make a deep copy in our constructor. I'm going to implement this change in the constructor:

```
private ImmutableSpaceship(String name, Destination destination) {
     this.name = name;
     this.destination = new Destination(destination);
}
```

Example 13

It is better to make this change inside the private constructor, because now, if we create other methods that modify the destination, they also will automatically make a deep copy of this mutable field.

Now, we've completely insured the immutability of our class:

```
public final class ImmutableSpaceship {

 private final String name;
 private final Destination destination;

 public ImmutableSpaceship(String name) {
       this.name = name;
       this.destination = new Destination("NONE");
 }

 private ImmutableSpaceship(String name, Destination destination) {
      this.name = name;
      this.destination = new Destination(destination);
 }

 public Destination currentDestination() {
       return new Destination(destination);
 }

 public ImmutableSpaceship newDestination(Destination newDestination) {
       return new ImmutableSpaceship(this.name, newDestination);
 }

[…]
}
```

Example 14

I hope you now have a better understanding of immutables, and can see just how useful they are. Remember to keep your code simple and clean, by using immutables to the greatest possible extent.

Thanks for reading!