**Core Object Oriented Development using Java**

OOD Week 1 – Module 13

Mutable & immutable classes

Tutorial

© FDM Group Ltd 2020. All Rights Reserved.

Any unauthorised reproduction or distribution in part  
or in whole will constitute an infringement of copyright.

# What does this tutorial cover?

This tutorial will introduce you to the idea of mutable and immutable objects. It will show you how to write your own immutable class.

# How long will the tutorial take to complete?

1 hour

# What should you have already completed?

Java core syntax, classes, objects, constructors, references, final keyword

# What do you need?

In order to complete this tutorial exercise you will need:

* Java Development Kit 1.8 or above
* Apache Maven
* Eclipse IDE Kepler or above

# What does this tutorial cover?

* Built in immutable classes – String
* Built in mutable classes – StringBuilder & StringBuffer
* Writing your own immutable class.

# Immutable objects

An immutable object is one where the values of its attributes can’t be changed after it’s been instantiated. Common built in examples in Java are the String class and all 8 primitive wrapper classes.

At this point, you may be thinking that a String is mutable, after all you could do this:

String string1 = "abc";

string1 = string1 + 'd';

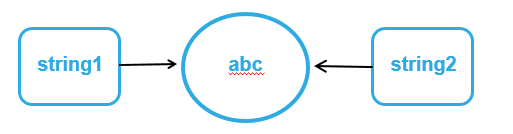
System.***out***.println(string1);

You’d see that the program prints out “abcd”. So it looks like string1 has changed. However, there’s a lot going on behind the scenes that we’re not seeing. Let’s try another example:

String string1 = "abc";

String string2 = string1;

In this case, the string1 and string2 references both point to the same String object containing the text “abc”:



But if we now make a change to string1:

string1 = string1 + "d";

System.***out***.println(string1);

System.***out***.println(string2);

We’ll see that string1 now has the value “abcd”, but string2 continues to have the value “abc”. Behind the scenes a new object has been created to store “abcd” and string1 has been re-pointed towards this object.



This proves that Strings are immutable.

## The String pool

One advantage of String immutability is that Java can save memory by pointing multiple String references at a single String object. For example:

String string1 = "abc";

String string2 = "abc";

System.***out***.println(string1==string2);

In this case string1==string2 will compute as true, because both references are automatically pointed to the same object. Provided we create Strings using the syntax above, there will only be one object per unique value stored in the memory.

If we create Strings using some alternative syntax, we’ll get a different result:

String string1 = **new** String("abc");

String string2 = **new** String("abc");

System.***out***.println(string1==string2);

In this case string1==string2 will compute as false, because each reference points to a different String object.

## Wrapper pools

Wrapper classes save memory in the same way as Strings by only having a single object per value stored in the memory.

However, you should be aware that this works in a much more limited way than with Strings:

Integer integer1 = 127;

Integer integer2 = 127;

System.***out***.println(integer1==integer2);

In this case integer1==integer2 will compute as true because both references point to the same object.

However if we try this:

Integer integer1 = 128;

Integer integer2 = 128;

System.***out***.println(integer1==integer2);

We get false. The reason is that only Integer values between -128 and 127 are stored in the pool. This makes sense as smaller numbers are most likely to be duplicated.

The same principle works for byte, short and long.

A downside of this is that if you use == to compare wrapper values in your code, it might appear to work when you’re testing with small values, but then start to fail when larger values are used. For this reason, you should always use the equals() method for comparing wrapper values.

# Mutable objects

A mutable object is one which has values that can be changed after the object has been instantiated. Every custom class that you’ve written up to this point has been mutable. Many built in Java classes such as ArrayList are also mutable.

In the previous section we saw how the immutability of Strings can save memory. Unfortunately, in some situations String immutability can waste a lot of memory. For instance if we have loop which changes a String’s value on each iteration, we could end up with a large number of discarded String objects littering the String pool.

## StringBuilder

This is a mutable alternative to using the String class. Like String it holds text. Unlike String the text can be changed without creating a new object.

Let’s test this out to prove the point:

StringBuilder sb1 = **new** StringBuilder("abc");

StringBuilder sb2 = sb1;

sb1.append('d');

System.***out***.println(sb1);

System.***out***.println(sb2);

This is the same example we tried with Strings earlier. The StringBuilder references sb1 and sb2 are initially pointed to the same StringBuilder object.

However when we append the letter ‘d’ to sb1, we find that both references now point to the value “abcd”. No new object has been created, we’ve just changed the contents of the existing object.

StringBuilder has lots of useful methods for manipulating text such as: append(), insert(), delete(), reverse()

Where we need to make lots of changes to some text, StringBuilder is a better option than String as it will save memory.

## StringBuffer

StringBuffer is very similar to StringBuilder. It’s mutable and has exactly the same methods as StringBuilder.

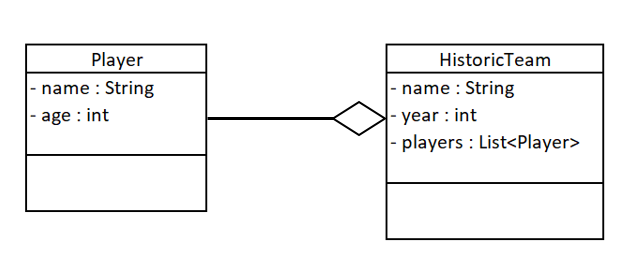
The big difference is that StringBuffer is ‘Thread Safe’. This means that it can only be accessed by one thread at a time. As we’ll see in OOD week 3, this could be very useful in a multi-threaded application. For instance if two threads tried to change the value of a StringBuilder object at the same time, we could end up with some corrupt data. This can never happen with StringBuffer.

On the other hand, StringBuilder is faster than StringBuffer. It should be used in situations where there’s no danger of two different threads trying to change its value at the same time.

# Writing an immutable class

You can find the completed code for this example in the code examples section on git.

We’re going to use the example in the UML below:



The HistoricTeam class will be an immutable class. It models something which can’t be changed, i.e. the past. The players might age, but in the historic team they will always be young!!

### Step 1 – make all the attributes of the immutable class final

This is probably the most obvious step. As we’ve seen before, final attributes are constant. They also don’t have setter methods. However, as we’ll see a bit later, this will only stop some of the historic team’s attributes from changing. Some attributes could still be changed even though they’re final.

### Step 2 – set all of the attributes through the constructor

As the final attributes don’t have setter methods, they need to be set through the constructor:

**public** HistoricTeam(String name, **int** year, List<Player> players) {

**this**.name = name;

**this**.year = year;

**this**.players = players;

}

### Step 3 – deep clone mutable objects in the constructor

At this stage, it would be easy to think that we’ve done enough to make our historic team class immutable. But we’re about to hit a problem.

Let’s make some player objects, add them to a List and then put them into a new HistoricTeam object:

Player player1 = **new** Player("player1",21);

Player player2 = **new** Player("player2",25);

Player player3 = **new** Player("player3",30);

List<Player> players = **new** ArrayList<>(Arrays.*asList*(player1,player2,player3));

HistoricTeam historicTeam = **new** HistoricTeam("US Hackathon Team",1985,players);

Now let’s change the ages of the player objects to reflect their ages today:

player1.setAge(56);

player2.setAge(60);

player3.setAge(65);

Finally, let’s check the ages of the players within the historic team:

List<Player> playersFromTeam = historicTeam.getPlayers();

**for** (Player player : playersFromTeam) {

System.***out***.println(player.getName()+" age: "+player.getAge());

}

When you run the code, you should see that the ages of the players within the historic team have been changed. At this point, it should be clear that the HistoricTeam class is not immutable.

This exposes a problem with Java. Each object in Java can have multiple references. Just because one of the references to the object is marked as final and private, doesn’t mean that the attributes within the object can’t be changed via one of the other references.

You can see another instance of this problem if you add or remove a player from the List which was passed into the historic team’s constructor. Doing this will also add or remove a player from the historic team.

The solution to this is to ensure that there are no external references to any of the mutable objects within the HistoricTeam class. We can do this by using a process called ‘deep cloning’. This involves making a copy of all mutable objects within our class. It also involves making a copy of mutable objects within mutable objects.

Let’s update our constructor to do a deep clone:

**private** **final** List<Player> players = **new** ArrayList<Player>();

**public** HistoricTeam(String name, **int** year, List<Player> players) {

**this**.name = name;

**this**.year = year;

**for** (Player player : players) {

Player clonedPlayer = **new** Player(player.getName(),player.getAge());

**this**.players.add(clonedPlayer);

}

}

Notice that we’re creating a new ArrayList of players within the HistoricTeam class. This guarantees that there will be no external references to the List.

Also we’re making a new Player object for each of the players in the original List. Again, this ensures that there will be no external references to any of the players.

### Step 4 – deep clone objects returned by the getter methods of mutable attributes

The issue with external references also applies to getter methods for mutable attributes of our HistoricTeam class. In this case if our getPlayers() method simply returns a reference to the ArrayList of players within the class, there’s a risk that the list could be modified outside of the class.

We’re going to use the same approach we used with the constructor. So getter methods will return cloned copies of mutable objects within the HistoricTeam class:

**public** List<Player> getPlayers() {

List<Player> clonedPlayers = **new** ArrayList<Player>();

**for** (Player player : players) {

Player clonedPlayer = **new** Player(player.getName(),player.getAge());

clonedPlayers.add(clonedPlayer);

}

**return** clonedPlayers;

}

### Step 5 – make the class final

At this stage it’s impossible to change any of the attributes within the HistoricTeam class. Unfortunately, we’re not quite finished! It’s still possible to make a mutable child class of HistoricTeam. This could then be passed into any method which required an HistoricTeam object as an argument.

Here’s an example of how this could be done:

**public** **class** MutableChildOfHistoricTeam **extends** HistoricTeam {

**private** String name;

**private** **int** year;

**private** List<Player> players;

**public** MutableChildOfHistoricTeam(String name, **int** year, List<Player> players) {

**super**(name, year, players);

**this**.setName(name);

**this**.setYear(year);

**this**.setPlayers(players);

}

**public** List<Player> getPlayers() {

**return** players;

}

// overridden getters & setters for all attributes

}

This approach is quite cynical. The call to super() in the constructor is only there to avoid a compile error. The 3 lines below it set non-final versions of the 3 attributes without using any cloning.

Preventing this sort of thing from happening is very easy, we simply make sure that our HistoricTeam class is a final class:

**public** **final** **class** HistoricTeam {}

### Summary

To make write an immutable class you must do all of the following:

1. Make all attributes final.
2. Set all attributes via the constructor.
3. Deep clone all mutable attributes passed into the constructor.
4. Deep clone all mutable attributes returned by getter methods.
5. Make the class final.