Immutability

Strings are immutable

You cannot change a String object once it exists

When we create a String object, either via the double quote mechanism with a literal String or via the operator `new`, we create an object representing a string of characters in the memory, just as with any other type of objects.

One thing which we must be aware of, though, is that such String objects cannot be changed once they are created. Even if we call an instance method on the string object like `toUpperCase()` which sounds as if we are changing the “contents” of the object, i.e. the character it represents, we are not changing anything.

All such instance methods in java.lang.String actually create a new object and return a reference to that new object.

How are strings represented internally?

The String class declares a private array of char primitives, which holds the characters the String object represent.

```
private final char value[];
```

The designers of the String class have decided that Strings are so common but also important, that it would actually be harmful if we allowed them to change their internal state.

What could go wrong if Strings were mutable

Strings can hold important data like the path to a file, a password, a user name, a class to be loaded or a URL to connect to.

If we allowed strings to be mutable (to be allowed to change their internal state), we would risk trusting a Strings which could later be changed when the program is running.

Consider if we had an accessor method to a mutable String. Then that String could be changed, even if the variable which is used internally is private.

```
SomeClass.getSensitiveValue().changeTo("malicious value");
```

Immutable Strings are safe to expose

A method returning a reference to a private String variable is still pretty safe, because no one can change the object's internal state, because Strings are immutable!

If Strings were not immutable, the following would have been possible:

```
SomeClass.PATH_TO_HOME_DIRECTORY.concat("../../some/other/");
```

If Strings were mutable, the concat method would just add the text to the end of the existing string in SomeClass! Luckily, Strings are immutable so we can't change any String anywhere. The above code would result in a new String.

Strings are re-used

Actually Strings created using the double quote technique are re-used. This would not have worked so well if Strings were not immutable.

```
String first = "ABC";

String second= "ABC"; // first and second refer to the same object!
```

If we could change the internal value of first, then second would be changed too, which means we couldn’t re-use the object!

Synchronizing benefits

There is actually lot of code which runs in parallell (we will not learn how to write such code in this book/course, however!). Having Strings immutable means that we don’t have to worry about synchronization problems if two pieces of code would have access to the same String at the same time, and one piece of code would change the state of the String without the other piece of code knowing about it.

If one part of the parallell program has access to a String, it can be sure that no other part will change that object, since it is not possible.

Should we make all objects immutable?

Some people actually promote this idea. It is possible to only create immutable objects in a system. This has to be documented and fully understood by the users of such objects.

It is, however, quite common to have mutable objects, so it is good to know and understand both concepts.

Sometimes objects can be partially mutable too. Think of a Member object which allows for its email value to change, but not the birthDate or name values. You have to decide for yourself and document your decision.

How do we make our classes immutable

An immutable class can't have any methods which change the internal representation of its state. A method which changes the representation should instead return a new object with the new value represented internally:

```
public Member changeEmail(String newEmail){
  return new Member(this.name, newEmail);
}
```

Also, any methods returning a value from an instance variable, can only return references to immutable types, such as String.

If we return a reference to a mutable instance variable, that variable can be changed, of course!