How Objects Can (or Can't) Change

When we apply methods such as `toUpperCase()` or `substring()` or even `concat(String)` to a `String`, we get a new `String` object returned. The original is unchanged.

However, we have written our own classes where the contents of an object *can* be changed via methods we create.
What about **Integer** objects?

The following two lines of code accomplish the same thing:

```java
Integer value1 = 8;
Integer value1 = new Integer(8);
```

The following line of code will cause `value1` to refer to an Integer object that contains 9:

```java
value1++; //This will unbox the int, increment it, and rebox it.
```

However, it’s NOT the same object as before!

```java
Integer value1 = 8;
Integer value2 = value1;
value1++; System.out.println(value1 + " " + value2);
```

---

**Integer** and **String** objects are immutable

It turns out that there are data types where once an object is created, it can never be altered. A copy based on it with some difference can be made, but the original is untouched.

In object oriented languages, some object types are immutable or *effectively* immutable.
Immutable Objects

With immutable objects, once the object is created, the values it holds cannot be altered.

With an effectively immutable object, the values it holds are not meant to be altered but could be using language tricks.

There are two big differences between immutable and effectively immutable in Java.

– First, true immutable object types need all data fields to be declared as final.
– Second, there are rules regarding the way the constructors work that need to be followed regarding them being thread-safe.

Mutable Objects

Object types where information is publicly available (and thus can change) or which provide public setters are called mutable.

Different languages approach mutability in different ways, so it is important to explore the conventions of each language you learn.

– In Objective-C there are usually both mutable and immutable versions of types.
– Though not an object, it's interesting to note that in Fortran, even literals like 1 weren't always immutable!
Shallow Copies

One place where we will need to think about whether an object is mutable is when we talk about aliasing and making copies of things.

Let’s consider a copy constructor and memory traces.

Our Student class contained:

```java
private String name;
private String ident;
private int tokenLevel;
```

What does the following copy constructor do in terms of memory?

```java
public Student(Student anotherStudent) {
    name = anotherStudent.name;
    ident = anotherStudent.ident;
    tokenLevel = anotherStudent.tokenLevel;
}
```

Shallow Copies and Mutable Objects

Image a class that contains references to two student objects:

```java
public class StudentPair {
    Student stu1;
    Student stu2;
    ...
}
```

Now imagine we had the following copy constructor:

```java
public StudentPair(StudentPair other) {
    this.stu1 = other.stu1;
    this.stu2 = other.stu2;
}
```

If we then had code such as:

```java
StudentPair sp2 = new StudentPair(sp1);
```

Where the new StudentPair created this way would contain aliases to the same Student objects as the original one. If sp1.stu1 had a token used, sp2.stu1 would as well!
Deep Copies and Mutable Objects

Image a class that contains references to two student objects:

```java
public class StudentPair {
    Student stu1;
    Student stu2;
    ...
}
```

Now imagine we had the following copy constructor:

```java
public StudentPair(StudentPair other) {
    this.stu1 = new Student(other.stu1);
    this.stu2 = new Student(other.stu2);
}
```

We could then have code such as:

```java
StudentPair sp2 = new StudentPair(sp1);
```

the new `StudentPair` created this way would contain a reference to different `Student` objects as the original one. If `sp1.stu1` had a token used, `sp2.stu1` would **NOT**!

What if we *want* mutable strings?

The `String` class in Java is immutable.
The `StringBuffer` class in Java is mutable.

Let's look at an example: `StringHolder`
**The StringBuffer Class**

Some key methods we can use are:
- append, which is overloaded in many ways
- insert, which is also overloaded in many ways
- delete, which allows you to delete any sub-part of the string
- replace, which allows you to replace any sub-part of the string with another string

These methods also return a **reference** to the `StringBuffer` being modified.

http://download.oracle.com/javase/7/docs/api/java/lang/StringBuffer.html

---

**Mutability: Good or Bad?**

While a significant issue, some might argue it's not "too bad" since aliasing of mutable objects can be "solved" by making deep copies when needed.

Are there any reasons why mutable objects would actually be good in their own right?

Let's look at an example:

`StringEfficiencyExample`
What are the trade-offs?

What are the trade-offs involved in the previous example?
- Time?
- Space?
- Other?

For which of the following are there aliasing risks?

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>