# More on Classes

toString

One "special" method in Java is the toString method. The method (regardless of which class it's added to) has the following prototype:

```
public String toString();
```

The job of this method is to return a String representation of the object. Here is the toString method from the Time class:

```
public String toString() {
    return (hours+" hours and "+minutes+" minutes");
}
```

In particular, let's say we define a Time object somewhere as follows:

```
Time t = new Time(75);
```

If we follow this statement with

```
System.out.println(t);
```

One's initial reaction might be, "how can we know how to print out a Time object?" The answer to this question is that whenever Java sees this, it automatically converts this line of code (without the user specifying) to the following:

```
System.out.println(t.toString());
```

Thus, in the context of a print or println statement, Java knows to replace an object with its String representation, which is defined by the toString method.

This begs the question, "what if I don't write a toString method in a class?" Every class the user creates (and every class in Java's API) inherits from the Object class automatically. This object class has a definition for the toString method. (We'll talk about inheritance later.) Thus, what happens if you forget to define a toString method is that the Object class's toString method gets called. This usually prints something like the following:

```
Time@1ef723
```

The first part of this output is the name of the class of the object. This is followed by the '@' character. The last six digits are hexadecimal digits (don't worry about what this means for now) that represent a location in memory where this Object is located.

Clearly, this isn't what we want to print out. Thus, to avoid this, you should define a toString method in every class you write.

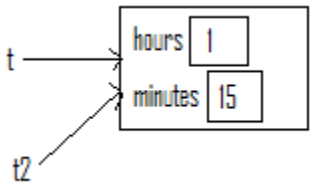
References versus Objects

In the pictures in the previous set of notes, a distinction was drawn between references and objects. References have names and objects don't. Thus, in those examples, t, t2, and t3 were all examples we have seen so far, the number of references available has equaled the number of objects. Because this is frequently the case, students often confuse references and objects as being one in the same.

The following example will illustrate the difference between the two:

```
Time t = new Time(75);
Time t2 = t;
```

After these two lines of code, the picture is as follows:




The key difference here is that there is only one object, but there are two references. The second line of code makes the reference t2 point to the same object that t points to.

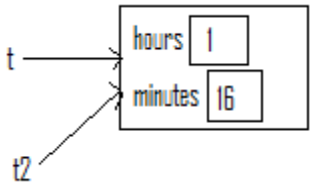
Now, imagine adding the following method to the Time class:

```
public void addMinute() {
    minutes += 1;
    if (minutes > 59) {
        minutes = 0;
        hours++;
    }
}
```

This method ***changes*** the object upon which it is called. Thus, if we made the following call:

```
t2.addMinute();
```

Our new picture would be as follows:



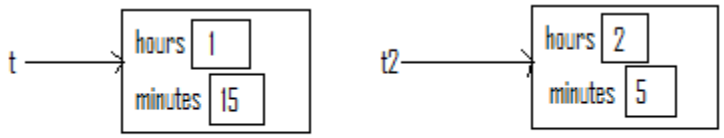

It's clearly evident from this picture that even though we didn't call any method on t, if we were to print out t, it would print out as "1 hours and 16 minutes".

Practical use of an Extra Reference

In certain instances, having more than one reference to the same object is what we want. Consider the situation where we want to swap two Time objects:

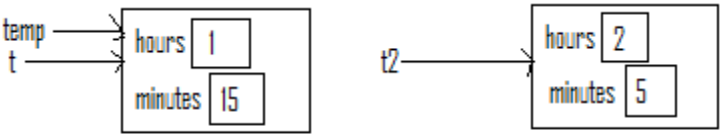
```
Time t = new Time(75);
Time t2 = new Time(125);
```

The picture after these two lines of code is:



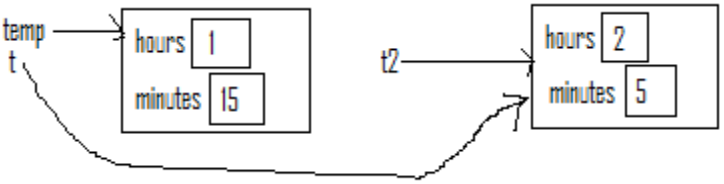

Let's say we want t to refer to the Time object that is 2 hours and 5 minutes and we want t2 to refer to the Time object that is 1 hour and 15 minutes. The truth is that we don't REALLY need a third Time object. We do, however, need a third Time reference:

```
Time temp = t;
```




Now, we are free to move the reference t:

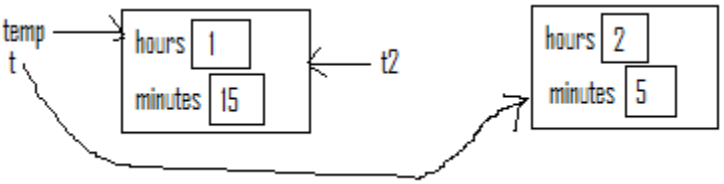
```
t = t2;
```




Notice how this line of code just MOVES the reference t to point to the same object that t2 points to, as was previously discussed. No new object is created here, but rather, a reference has moved.

Now, we conclude with the line:

```
t2 = temp;
```




The swap is complete and we never had to create a third object!

Cloning

Other times, however, we will want to create a real copy of an object, not just have two separate references pointing to the same object. The way to do this is through what is often called a "copy constructor." (This terminology is usually only used in describing the same function in the language C++.) This simply means creating a constructor that takes in an object from the class. Here is a copy constructor for the Time class:

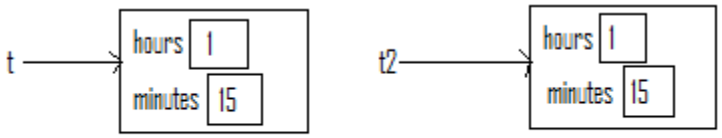
```
public Time(Time time2) {
    hours = time2.hours;
    minutes = time2.minutes;
}
```

It should be fairly obvious that what this constructor does is take in a Time object and simply copy each of its components into the new object being created by the constructor.

Now, consider these two lines of code:

```
Time t = new Time(75);
Time t2 = new Time(t);
```

The corresponding picture is as follows:




Here, t2 is NOT referring to the same object as t.

CD Class Example

Consider a CD object (these were popular in the 1990s and part of the 200s). For our purposes, a CD will have four instance variables:

```
private int sku;
private String artist;
private double cost;
private String title;
```

A sku is a unique identifier (nowadays a UPC) and the others are typically associated with a CD object. If someone is going to create a CD object, we would like to provide them some functionality, but not the ability to change things about the object in any random way.

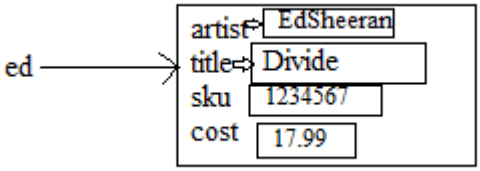
As mentioned in a prior lecture, Java's String class is designed so that once a String object is created, no changes can be made to the object. Instead of making changes to the object, most of the Java string methods create a new object that reflect some change made to an old object. The CD class has a method like this. Many artists like to put out sequels to their past work. Given this proclivity, consider the following method:

```
public CD makeSequel() {
    return (new CD(artist, title+"II",sku+1,cost+2));
}
```

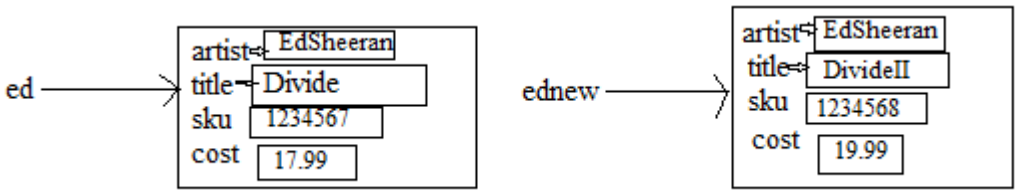
This method does NOT change the current object (this), but rather creates a new one and returns it! Here is a quick picture of a situation with the following lines of code:

```
CD ed = new CD("EdSheeran","Divide",1234567,17.99);
CD ednew = ed.makeSequel();
```

After the first line of code we have:




and after the second line of code we have:

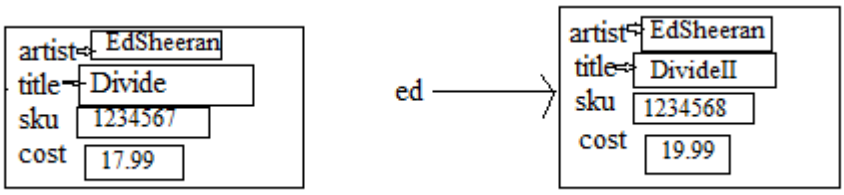



Thus, in this instance, no change was made to the object pointed by ed, but a new object was created in relation to it with some modifications.

It's common to provide methods that create new objects with changes without changing the original object. While coding everything like this makes for some inefficiencies, it typically leads to less buggy code and code that takes less time to debug. If someone has no need for the newly created object, one can just reassign the original reference to it. For example, we could change the second line of code to:

```
ed = ed.makeSequel();
```

If we did this for our second line instead, our picture would be:




What would happen shortly thereafter is that the object on the left, which doesn't have any references pointing to it, would get garbage collected.

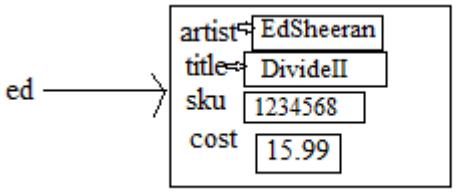
A different approach would be to provide a method that can "mutate" or change the object itself that a reference is pointing to. Here is a method that does exactly this for the CD class:

```
public void discount(double rate) {
    cost-=cost*rate;
}
```

Consider running this line of code right after the picture that is directly above:

```
ed.discount(.2)
```

The new picture would be:




Thus, when designing a class, you must take into account what sort of functionality you would like to provide the user. Would you like to provide methods that change the object itself, provide methods that create new objects based on previous ones, or a mixture of both. The key is that you should provide the user reasonable functionality without allow the user to make undesirable changes to the object. Thus, the public methods limit the possible behaviors of the object. But from there, the user is free to combine/use these methods in any way she sees fit.