# Lab 9: Exceptions, Cloning, Copying, and Equality

Dale Vaillancourt

May 19, 2003

## 1 Exceptions

What is the value of `3 / 0`? What happens when you ask Java to compute it for you? Find out by running the provided `DivideByZero()` test. See that error message printed in the console? The JPT designers thought ahead about what happens when errors such as these happen, and they catch them in order to give you useful feedback. It's important to know how to catch these errors in order to make your applications handle failure in an appropriate manner.

We're going to learn the basics in just a moment, but first here's some vocabulary that every respectable computer scientist and software engineer should be familiar with:

- Exception - a type of value that indicates an error condition. For example, division by zero causes an ArithmeticException.
- Throw an exception - to signal programmatically that an error condition is present. (Some languages use the keyword `raise` instead of `throw`.)
- Catch an exception - to trap a thrown exception and perform some appropriate action. (Some languages use `handle` instead of `catch`.)

Okay, let's get to it. In Java, exception handling looks like this:

```
... some program text ...
try {
 ... statements that might cause errors go in here...
 } catch (Exception e) {
 ... statements that handle the Exception go here ...
 }
 ... more program text ...
```

Let's build a concrete example.

1. Open up `TestSuite.java`.
2. Find the `DivideByZero()` method.
3. Place the the offending statement inside a `try`-block.
4. Write a `catch`-block which prints a helpful error message to the console.

Note that by using `Exception e` in our `catch`-block, we state that we will handle any kind of error. This is analogous to saying `Object x = new ...;`. `x` can hold any kind of `Object`, and we don't really care what kind it is. In future courses you'll learn all about the different kinds of exceptions, and you'll be able to set up `catch`-blocks to handle certain kinds of exceptions and ignore others.

As a final note, uncaught exceptions will cause your program to crash. Remember the last time you stayed up late trying to finish that paper, and your XYZ brand whiz-bang word processor crashed before you could save (or worse, while you were trying to save)? That's an uncaught exception. Some programmer said to him- or herself "Nah, that'll never happen." and ignored a possible error condition. Needless to say, there is a lot of bad karma associated with such decisions.

## 2 Cloning

You've got an object, and you'd like to perform some computation on it. Unfortunately that computation involves destructive updates, and you want to keep the original object around when you're done with the computation. In this situation you need to make a clone of your object before you run the destructive computation on it. Say I've got a couple of objects:

```
Balloon b1 = new Balloon(10, 5, 10, Color.blue);
Balloon b2 = b1;
```

And I then want to change `b2`'s color:

```
b2.color = Color.red;
```

The problem is that `b1` also sees this change; `b2` does not contain a copy (or a *clone*) of `b1`, it actually references the same object. In order to clone objects, your class must implement the interface `Cloneable`. This interface requires that your class implement one method: `public Object clone();`. Study the `clone()` method in the `Balloon` class; the process of writing the method is entirely mechanical. It makes a copy of each field and creates a new object with the copies.

To do:

1. Try the `TestBalloonView()` test. Try moving each of the Balloons. What's happening with the red one?
2. Look at the `TestBalloonView()` code. Fix it to actually create two separate copies of the red balloon using the `clone` method.

## 3 Equality

Sometimes you've got two distinct objects, and you'd like to know if they are equal. That is to say, you want to know if the two objects you've got represent the same piece of information. For example, say I've defined these three `String`s:

```
String s1 = "Viera";
String s2 = "John";
String s3 = "Viera";
```

Obviously `s1` and `s3` are equal (even though they are two distinct objects), `s1` and `s2` are not equal, and `s2` and `s3` are not equal. Wouldn't it be nice to be able to determine this programmatically? Java's `String` class allows you to compare two `String`s for equality with the `equals` method. For example:

`s1.equals(s3);` evaluates to `true`
`s1.equals(s2);` evaluates to `false`

In fact, Java's class `Object` (which *every* class implicitly extends) provides a default `equals` method whose signature looks like this: `boolean equals(Object o);`, but its default implementation does not do what you want! The default implementation will only return `true` when this object and the given object are actually the *same object*. The `String` class is able to compare two different objects and tell you the Right Thing (TM) because it provides a smarter implementation of the `equals` method. To compare `String`s for equality, you just compare them character by character. If they all match, the two `String`s are equal. Let's use this idea to compare Balloons for equality!

Your job is to implement `equals` for Balloons:

1. Run `TestBalloonEqual()` and look at the output. Notice what's going on in the third test case. If you don't understand what's going on in this case, ask a TA or a tutor!
2. Open the file `Balloon.java`
3. Define the `equals` method with the signature given above.
4. Use `TestBalloonEqual()` to test your code. The output in the third case should make sense now.