Initializing Classes and Constructors

Initializing member variables

Member variables will be given default values on instantiation. For numbers, the default value is 0, for Booleans it is false, and for object references it is null.

However, there will be times when you want to override these default values when instantiating objects. For example, with the `ClickerCounter` class we've been developing so far:

ClickerCounter.java

```
public class ClickerCounter {

    // Member variables
    private int count;
    private int maxCount;

    // Accessors and mutators
    public int getCount() {
        return count;
    }

    public void setMaxCount(int maxCount) {
        if (maxCount > 0) {
            this.maxCount = maxCount;
        } else {
            System.out.println("Invalid maximum count: " + maxCount);
        }
    }

    // Interface methods
    public void click() {
        if (count < maxCount) {
            count++;
        } else {
            count = 0;
        }
    }

    public void reset() {
        count = 0;
    }
```

```
}
```

upon instantiation, the `maxCount` member variable will be initialized to zero. Given that a counter that counts from zero to zero isn't very useful, it would be good if this were automatically initialized to some other value.

One way to do this is to set initial values in the variable declarations:

```
public class ClickerCounter {

    // Member variables
    private int count = 0;
    private int maxCount = 9999;

    ...
}
```

With the above modification, when a `ClickerCounter` is instantiated, its `maxCount` will be set to 9999. We also explicitly initialized `count` to zero so that someone reading the code would be sure that the member variable gets the initialized with the value we wanted rather than wondering whether we forgot about initializing it.

This approach works fine if your class is fairly simple.

Default constructors

Another way to initialize a class is by using a **constructor**: a method that runs automatically whenever you instantiate the object. Constructors are better suited to more complex situations, and they are not limited to just initializing member variables. You can do anything in a constructor that you can do in a regular method.

For example, after initializing the state of your object, you might want to output that your object was successfully created. A constructor that does does this looks like:

```
public class ClickerCounter {

    ...

    // Default constructor
    public ClickerCounter() {
        // initialize member variables
        maxCount = 9999;
        count = 0;
```

```
        // output success
        System.out.print("Successfully created a ClickerCounter ");
        System.out.print("with a maxCount of " + maxCount + '.');
    }

    ...
}
```

Note the syntax. A constructor:

- does not have a return type (not even `void`).
- has the same name as the class.
- is declared with `public` access.

Now when we instantiate a `ClickerCounter`:

```
var myClicker = new ClickerCounter();
```

`myClicker`'s `maxCount` will be set to 9999, `count` will be set to zero, and a message indicating successful instantiation will be printed.

In short, constructors are used to initialize objects and can do so in ways that go beyond simple member variable initialization.

Parameterized constructors

The above is a **default constructor** because it has no parameters. It's also possible to define constructors that have parameters. Such a constructor is called a **parameterized constructor**:

```
public class ClickerCounter {

    ...

    // Parameterized constructor
    public ClickerCounter(int maxCount) {
        // initialize member variables
        this.maxCount = maxCount;
        count = 0;

        // output success
        System.out.print("Successfully created a ClickerCounter ");
        System.out.print("with a maxCount of " + maxCount + '.');
    }

    ...
```

```
}
```

You can define as many constructors as you want. You are not required to define a constructor, but if you define constructors, you must always define a default constructor.

toString()

One thing you'll want to get into the habit of doing with your class definitions is to write a `toString()` method. The method should return a string representation of the object. What you want that representation is up to you. The `toString()` method will automatically be invoked whenever Java needs a string representation of your object. An example of one for the `ClickerCounter` class might look like:

```
public String toString() {
    return "count: " + count;
}
```

Now if we want to output the state of a counter using `System,out.println()`, we could simply write:

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

Writing a `toString()` instance method is especially important when your class is more complicated than the simple class we've been using here. To understand the magic by which this works, you need to have an understanding of inheritance in Java. But that shouldn't stop you from doing this now.

equals(other)

Another method that you can define and is a good habit to get into defining is `equals`. A very basic definition of `equals` is created automatically, but in almost all cases, you will want to change this functionality. An example of an `equals` method for the `ClickerCounter` class might look like:

```
public boolean equals(ClickerCounter other) {
    return (this.count == other.count &&
            this.maxCount == other.maxCount);
}
```

It's up to you to decide what “equals” means. In many cases, “equals” means that two objects of the same class have the same state. So, in the above definition, for two counter to be equal, both the count and the maxCount must be the same. Now to use this method, assuming both of the variables below refer to valid `ClickerCounters`:

```
if (myCounter.equals(yourCounter) {
    System.out.println("Our counters are the same.");
} else {
```

```
    System.out.println("Your counter and mine are not the same.");
}
```