Function toString

Function toString for class Chapter

In the DrJava right upper pane, you see class `Chapter`, with three fields: the chapter number, chapter title, and the previous chapter. Along with two constructors and getter/setter methods. We often want to be able to get a description of an object of such a class —we want to see the values of the fields in a nice format, to make clear which value is which.

The Java convention for obtaining a description of an object of a class is to write a function `toString` in the class. Here is a standard specification for it, although one might go into more detail in some cases. It's **public**, it returns a `String`, its name is `toString`, and it has no parameters. We insert a return-statement, so that the class can be compiled, and compile.

```
/** =  a representation of this chapter —for the previous
       chapter, the chapter title is given. */
public String toString () {
    return "";
}
```

Now let's write the real return statement. Let's start off with the word *Chapter*; catenate to it the chapter number, a period, and a blank; and catenate to that the title and another period and blank. For the previous chapter, we have to be careful. We write a conditional expression; if the previous title is null, we append "No previous chapter", and if it is not null, we append the title of the previous chapter, as required by the spec.

```
return "Chapter " + number + ". " + title + ". " +
       (prev == null ? "No previous chapter"
                     : "Previous chapter title: " + prev.title);
```

Now let's check it out. We create two objects of class `Chapter` —let's make sure one of them has a non-null previous field—and call their `toString` functions. See?

```
Chapter c1= new Chapter(1, "Intro");
Chapter c2= new Chapter(2, "Truth", c1);
```

And that's all there is to writing a function `toString`. Generally, the result contains all the fields, but there are times that that will be too much, and a less complete `toString` function will be satisfactory.

Funtion toString for class Point

We write a function `toString` for class `Point`, an instance of which contains a point `(x, y)` in the plane —`x` is the horizontal coordinate and `y` the vertical coordinate. We have the two fields, `x` and `y`, and a constructor, which allows both fields to be initialized in a new-expression. We haven't written other methods because they are not needed for this demoinatration.

Here comes function `toString`. First, we put in a specification for it; note how the spec tells us what the represention will be. The function is public; it returns a `String`; its name is `toString`, and it has no parameters. We put in a return statement in so that we can compile.

```
/** = a representation "(x, y)" of this point */
 public String toString() {
     return "";
 }
```

We write the real return statement. We start with the opening parenthesis; catenate field to it; catenate to that a , and blank; then field y, and finally the closing parenthesis.

```
return "(" + x + ", " + y + ")";
```

That wasn't hard! Let's construct a few objects of class Point and see what their toString function produces.

```
(new Point(4, 5)).toString()
(new Point(0, -3.2)).toString()
```

Function toString

Automatic use of function toString

Suppose an object appears as an operand of a catenation, like this:

```
p= new Point(3, 5);
"Here is the point: " + p
```

In this situation, function `p.toString` will be used in place of `p`, as you can see when Drjava evaluates this expression. This allows us to abbreviate and keep things simple.

There is another reason for writing `p` instead of `p.toString` in these cases. If `p` contains **`null`**, evaluation of `p.toString` in

```
"Here is the point: " + p.toString()
```

causes an error, but evaluation of `p` in

```
"Here is the point: " + p
```

does not —the value **`null`** is simply changed into a `String`.

Another place where toString is not necessary is in a println statement, as you can see from execution of this statement:

```
System.out.println("A point: " + new Point(1,2));
```

Conclusion

The Java toString convention is built into the language, in that in several places, the use of an object will automatically result in its function `toString` being called. Make heavy use of the Java toString convention. Further, whenever you write a new class, immediately after writing the constructors, write function `toString`! Doing so will save you testing and debugging time, because you will be able to look at the descriptions of the various objects that your program creates.