Specifications of methods

You know what the first function `mini` does, because its specification, in the comment preceding it, tells you. You don't need the function body —which we show you now.

You have no idea what the second function does, because its specification has not been given. To understand what it does, you would have to look at its body: `Integer.signnum`.

A specification of a method defines what a call on a method does. The specification should be precise, complete, and as clear as possible. So that you don't have to look at the method body.

In fact, *every* method should have a specification, and you should *never* have to look at the method body when writing a call on a method. The method body indicates *how* a task is carried out. When writing a call on a method, we don't want to know that, we want to know *what* the task is. And the spec should tell us.

Writing procedure specifications

We write method specs as *Javadoc comments* immediately preceding the method. Javadoc comments begin with `/**` and end with `*/`. The next lecture will explain why we use Javadoc comments.

Let's begin discussing specs for procedures. There are two points to make here.

```
/** Set the title to t */
public void setTitle(String title){...}
```

1. The spec mentions every parameter by name, and in a way that explains what that parameter is for. If a spec doesn't mention a parameter, the reader may not understand what the parameter is for, and the spec is incomplete.

2. This spec is a *command* to do something. We don't say, "this procedures sets the title to t", or something similar. We say, "Set the title to t." This is because we use it as follows. Suppose we have a call on procedure `setTitle`: `setTitle("I want peace");` . The way to figure out what it means is to replace it by the specification, but substituting the argument for the parameter:

```
Set the title to "I want peace".
```

Since the procedure call is a command to do something, so should be the specification.

Writing function specifications

Since a function returns a value, the spec should be a description of that value. In fact, it should say what a call on the function equals. For example, that's what the spec of function `areAscending` does. It says that a call `areAscending(a, b, c)` is equal to the value of the true-false sentence "`a`, `b`, and `c` are in ascending order."

```
/** = "a, b, and c are in ascending order".*/
public static boolean areAscending(int a, int b, int c){...}
```

Since functions evaluate to a value, we often write their specifications starting with the equals sign. You may see a spec like "Return true if a, b, and c are in ascending order." This has a sort of operational feel, which we would rather not have. In mathematics, if someone asks, "what is sin(0)," one doesn't say, "return the value of the sin of 0,", one says simply," that's (equal to) the sin of 0. Same thing with any function call.

Writing constructor specifications

Here's how we write a specification for a constructor. The word "Constructor" announces that is is a constructor. And it explains what the properties of the new instance —what each field contains. Notice that the spec says what the previous chapter will be, even though there is no parameter for it. Note also that all parameters are explained:

```
/** Constructor: a new instance with chapter number n, chapter title t, and previous chapter null. */
public Chapter(int n, String t) {...}
```

Good specifications

Specifications of methods

Get in the habit of writing a method specification *before* you write the method body. For two reasons. First, you won't forget to write the spec, Second, that spec will help *you* the programmer, as you write the method body. You will be able to refer to the specification as you write the body. You don't have to remember everything about the spec.

Make sure that your spec is accurate and complete. It must be consistent with the method body. A spec that is wrong in any way is probably worse than no spec at all, because a programmer who relies on the specification in writing a call on the method is bound to be introducing an error into their program. [Adam: do NOT say "his or her". Say "their". After reading this, delete this comment.]

Finally, include explicitly any *preconditions* on a call —that is, constraints that the caller must satisfy. For example, the specification of a function that computes the square root of r should include the statement:

> Precondition: r >= 0.

Such a precondition need not be checked the function; it is up to the caller to call the function only when the precondition is true.

Changing a spec

If you realize you need to change what a method does, first change the specification, and *then* change the method body. Be disciplined in doing this, and you will make fewer errors.

Let's show how this works. Here's our first step: changing the specification.

And here's our earlier function `areAscending`, which may not treat equal parameters the way we want. For example, if a = b < c, true, and not false, should be returned. So, we change the spec to:

```
/** = "a, b, c are in non-descending order (a <= b <= c)" */
```

We have revised the spec. The next step is to revise the function body to be consistent with the spec, and here it is.