Static methods and variables

Setting the scene

Here's a version of class `Chapter`, with three fields: the chapter number, the chapter title, and the previous chapter. There are the usual constructor and getter methods and setter methods —we don't show them all.

Here is the file drawer for class `Chapter`, showing two objects of the class. Again, not all methods are shown in the objects.

We also declare a function `isBefore`, which tells whether the object in which it appears comes before chapter `c`. And the new function appears in all objects of class `Chapter`. There is an important reason for the function to be in each object. Each method body contains a reference to field `number`. In object `a0`, the reference to `number` accesses field `number` in `a0`. In `a1`, the reference to `number` accesses field `number` in `a1`. This is a consequence of the inside-out rule, which we discuss in detail in the next blecture.

A function that references no components

Here's another function with the same intent, to see whether one chapter comes before another one. In this function, both chapters are given as parameters. We now have two functions with the same name; the name `isBefore` is overloaded.

This function appears in every object of the class. However, it doesn't reference any fields and doesn't call any methods in the class. So why should this function appear in every object? There is no need for that.

We can delete this function from the objects by giving it attribute **`static`**. Now, a *single* copy of it is in the class file drawer. That's right:

> **Rule**: A method that is declared static appears in the file drawer —and *not* in each object.
> There is only one copy of the method.

We can't make the first function `isBefore` static because its reference to number is now illegal —there is no variable `number` in sight. Here, we'll show you how Java complains when we make that function static and attempt to compile.

So, we take off the static attribute. And, we have the following guideline:

> **Guideline**: Make a function or procedure static if its body does not refer to a field or instance method of the class.

Suppose you want to use only one function `isBefore`. Should you use the nonstatic version or the static version? We can't really answer that question now. The answer depends on the class being defined and how one is expected to use the class, and this is not the topic of this discussion. Here, we focus only on the mechanics of defining static methods and the consequences.

Class variables

One can also declare static variables in a class. To show this, we declare a variable `numberOfChaps`, which is to contain the number of objects of class `Chapter` that have been created thus far.

```
// no. of chapters created so far.
private static int numberOfChaps= 0;
```

Such a variable appears directly in the file drawer. And there is only one copy of it. It is created just before *any* static component is referenced or the first instance of the class is created. It is initialized either to the default value for its type or to the value given in an initializing declaration.

A static variable is also called a *class variable*, because it belongs to the class, or file drawer, and not with each object of the class.

When should a variable be made static? When only one copy of the variable is needed, for example, when it accumulates information about all objects in some way, or communicates information about all objects in some way. Class variable `numberOfChaps` is an example of this. It wouldn't do to have this variable in all objects.

Maintaining class variable numberOfChaps

Class variable `numberOfChaps` is supposed to contain the number of objects of class `Chapter` that have been created. The simplest way to maintain the variable is to increase it by 1 whenever an object is created, and that indicates that it should be increased in the constructor —because the constructor is called whenever a new object is created.

```
numberOfChaps= numberOfChaps + 1;
```