Field declarations and getter/setter methods

Consider writing a class `Chapter`. Each instance of `Chapter` will contain information about a chapter of a book, like the chapter number, so we need to declare a variable in the class. The basic form of a declaration of this variable, as you know, is **`int`** `number`, but we put a semicolon after it:

```
int number; // The chapter number.
```

This is what an object of class `Chapter` now looks like —yes, it is initialized to the default value 0.

A variable declared in a class definition is called a *field* or *instance variable*, because it is appears in every instance of the class. And, you can see that we can create an instance of the class, assign a value to the field, and tobtain the value:

```
x= new Chapter();
x.number= 3;
x.number
```

Access modifiers for fields

Generally, we give an *access modifier* in the declaration. Access modifier **`public`** allows access to the field from any place in which the instance is accessible. Instead, we use access modifier **`private`**, which signifies that the field is accessible only from code within class `Chapter`.

Here, we'll show you. Compile the program, create an instance of it, as before, and then try to assign field `number` a value —Java complains; it is not allowed. Even an attempt to obtain the value of the field leads to an error message: IllegalAccessException: cannot access a member of class `Chapter` with modifier **`private`**.

In software engineering, we use the general rule that fields of a class are **`private`**. There will be exceptions to this rule, but from now on, make fields of classes **`private`**.

Let's also add a declaration of a field to contain the title of the chapter —notice how we put a comment that gives the meaning of the variable. The default value in a field of a class-type is **`null`**.

```
private String title; // The title of the chapter.
```

Getter functions

We have a problem: we can't look at or change the fields, so what good are they? To solve this problem, we write two kinds of methods.

First, we introduce *getter* methods, or functions, to *get* the value of the fields. Note how we start off with a specification, which says what value the function returns. And we use the standard Java convention that the name of the function is the name of the field preceded by the word `get`. We compile, to make sure there are no syntax errors. We compile *very* often. Let's put in a getter function for field `title`, too —we'll do it by copying the first, pasting, and editing. We compile.

Finally, we use the interactions pane to show that we can now access the the fields of an instance using he getter functions.

Setter procedures

To be able to store values in the fields, we define so-called *setter* methods, or procedures. We put in a specification for a procedure to store a value in field `number` —note that it is a command to do something. We put in the header of the procedure: it's **`public`**; it's a procedure, so we use keyword **`void`**; the name is the word `set` followed by the name of the field; and we put a *parameter declaration* within parentheses to indicate the variable, `n`, that will contain the value to be stored in field `number`; and we end with the braces for the procedure body. Compile to make sure we haven't made a syntactic error.

Finally, we put an assignment in the body to assign parameter `n` to field `number`. And compile. And now we show in the interactions pane that we can use the setter procedure to assign a value to the field and then use a gett function to retrieve the value.

To complete our work, we put in a setter procedure for field `title`.

Field declarations and getter/setter methods

About parameters and arguments

This is the second method you see that has a parameter. Later, we will go into more detail about how an argument value is associated with the parameter when the method is called. For now, we slip by this issue, assuming that you have seen function, procedures, and calls on them in another programming language.

Discussion

We made a field **`private`** so that it couldn't be accessed. We then added a getter method and setter method to provide that access! Later, we will see the software-engineering reasons for allowing access in this fashion. You don't *have* to provide either the getter method or the setter method. If you don't want to allow access to a field, don't provide a getter method. If you don't want to allow someone to change a field, don't provide a setter method.

So, for now, accept that fields should be made **`private`** and write getter/setter methods if you want.