The class definition

We write our first Java class, in the upper right pane of DrJava. This class definition will describe the components that are placed in each manilla folder, or object, of the class. We start by putting a comment that indicates what each instance of the class has in it.

```
/** An instance is like a JFrame but has two more methods */
```

We then type **`public`**, which means that this class can be referenced from anywhere else; **`class`**, which means that we are writing a new class definition; a name for the class, we choose `Demo`; and an opening and closing brace. The opening brace goes on the same line with the name of the class.

The words **`public`** and **`class`** appear in blue, because they are keywords and cannot be used as identifiers —as names of variables, methods, or classes.

Instances of class Demo have essentially nothing in them, because we haven't declared any components.

We save this class definition, by clicking button `Save`. A navigation window opens, and we navigate to where we want to place the program, create a new folder, and save the program with the same name as the class. *Every program, consisting of a bunch of classes, should be placed in its own folder*!

The name of the file in which this class definition was saved, `Demo.java`, now appears in the left pane. And we also see file `Demo.java` in the righthand pane.

Here is a rule to follow:

```
/* Rule for file names: Each public class definition should be
   placed in its own file; the name of the file should be name
   of the class, and the extension should be java. */
```

Let's look in the folder to see whether the file is there. Yes, there it is.

Compiling the class definition

Now, we compile the class definition, by clicking button `Compile`. This translates the class definition into a machine language, so that it can be executed. Look in the folder that contains `Demo.java`. There is now a file `Demo.class` —this file contains the machine language version of class `Demo`.

Creating objects of the class

We can now create an instance of the class, by typing:

```
j= new Demo();
```

in the interactions pane. Notice the use of a new-expression to create an object of class `Demo`. But there are no methods in it, as far as we know, because we did not define any in the class definition. For example, method `show` does not exist, which we can verify by typing `j.show()` into the interactions pane.

However, we can ask for the value of expression `j` —its value is the name on the tab of the manila folder! It consists of the name of the class, an @ sign, and then some other characters.

Extending class JFrame

Now, we would like instances of class `Demo` to have all the fields and methods that an instance of class `JFrame` has. We do this in two steps. First, we put an *import statement* before the class definition:

```
import javax.swing.*;
```

This import statement indicates that all the classes in package `javax.swing` may be used.

Next, after the class name, we put an *extends-clause*, indicating that class `JFrame` is to be extended. Note that **`extends`** is a keyword. We compile the program again and then execute the assignment of a new instance of class `Demo` to `j`. Now, when we execute `j.show();`, the `JFrame` window appears on the screen! This is evidence that the use of the extends-clause has caused all the components of `JFrame` to be *inherited* —they

actually appear in each folder of class `Demo`. We can set the title of the window using procedure `setTitle`, and we can get the width of the window —we can use all the methods that appear in every `JFrame` object.

In the next lecture, we show how to draw a manilla folder of class `Demo`, and we show how to define a function and a procedure in class `Demo`.