The Development Environment

L2.1 What Is a Development Environment?

A development environment is made up of all the tools and documents needed to develop applications. It consists of six elements; the first three are platform-independent, and the last three are not.

A Working Directory This is where all your programs are stored.

API You will need easy and quick access to the API of Java and TYPE.

TYPE This refers to the family of programs and packages that come with this textbook. It is bundled in one *type.jar* file that installs as an extension of the standard library.

The JDK The JDK for your particular platform must be installed.

An Editor An editor is a program that allows you to create, modify, and save source programs.

A Console A console is a window through which you can run, and interact with, your programs.

L2.2 Setting Up the Development Environment

You can find everything you need to set up a development environment on your home computer on the book's web site:

```
http://www.cse.yorku.ca/~roumani/jba
```

There is a separate section for each operating system. Read the one that pertains to the O/S of the machine on which you want to install the environment.

L2.3 Testing the Environment

1. Launch the editor and the console.
2. In the editor, open the file *Demo.java* in your working directory.
3. Compile it in the console.
4. Verify that *Demo.class* was created.
5. Run the created class.

If the above test failed, you should examine the reported error message and determine the source of the problem accordingly. Visit the book's web site for help.

L2.4

Open the console.
Issue the command *java Options*.
Set your login data.
Set the eCheck URL given by your instructor.
Click *Apply*.

eCheck uses the above data to communicate with a server at your department and record that you have successfully completed assigned programs. If your course does not use the eCheck server, or if you want to eCheck your programs off-line, click the `"Offline"` check box. This way, you can still benefit from eCheck as a learning tool.

L2.5

We like to write a program that figures out how long it takes for the end-user to respond to a prompt. Specifically, it prompts the user to press ENTER and outputs the number of milliseconds that have elapsed between printing the prompt and receiving the input.

Start by creating a program based on the template of Fig. 2.14. It is a good idea to save the file under the name `Template.java`, so that you can reuse it in future projects, and then to resave it (using the `save as` menu) under the name `Lab2_1.java`.

We will delegate keeping track of time to a class in `java.lang` called `System`. One of its methods is static and has the following API:

```
long currentTimeMillis()
```

Returns: the difference, measured in milliseconds, between the current time and midnight, January 1, 1970.

The header indicates that the method returns a `long` value and that it takes no parameters (evidenced by the empty parenthesis pair in the signature). Since this method is static, we invoke it on the class as in the following example:

```
long now = System.currentTimeMillis();
```

To measure a time interval, we invoke the method just before and just after the interval, and then subtract the two returns. Here is a template:

```
output.println("Press ENTER");
long start = System.currentTimeMillis();
input.nextLine();
...
```

We used the `nextLine()` method because it can read any entry, including an empty one (generated when the user presses ENTER without typing anything). Notice that we trashed the return (rather than store it in a variable) because we are not interested in it, only that it took

place. Continue the development and create the class Lab2_1 such that its I/O is similar to th following sample run:

```
Press ENTER
Pressing ENTER took you 443 milliseconds.
```

In particular, your output must have one space before and after the shown duration.

Note: the nanoTime() method was added to the System class in Java 5.0. It is similar to the one used above, but it is more precise because it returns the time in nanoseconds.

L2.6 Approximating Today's Date

We saw in Section 2.1.3 that one can use the Date class to obtain today's date, and we will se in Chapter 7 that one can use the Calendar class to express a date in various locales and tim zones. In this section, and as an exercise, we will figure out today's date from the number c milliseconds between January 1, 1970 and now. We will ignore leap years and assume 365 day per year.

To avoid magic numbers (which would be a style violation), our program starts by nam ing a couple of constants:

```
final int BASE_YEAR = 1970;
final long MS_PER_YEAR = 1000 * 3600 * 24 * 365;
```

As a start, let us compute and display today's year:

```
long now = System.currentTimeMillis();
long year = now / MS_PER_YEAR;
output.println(BASE_YEAR + year);
```

Create the program Lab2_2, compile it, and run. Did you get the result you expect? Why not Our program does in fact have a logic error in it. Spend some time debugging it so that yo can determine what is wrong. The best debugging tool is printing: simply print each and ever intermediate value that your program uses. Ironically, *the error can be fixed by inserting on character only.*

After correcting the problem, and obtaining the correct year, continue the developmen so that today's month and day are also displayed. You may want to add two more constants:

```
final long MS_PER_MONTH = MS_PER_YEAR / 12;
final long MS_PER_DAY = MS_PER_YEAR / 365;
```

You should benefit from the built-in remainder operator. For example,

```
long left = now % MS_PER_YEAR;
```

yields the number of milliseconds that remain left over after the years are accounted for. Note that since leap years were ignored, it should not be surprising if the computed date is off by 10 days or so. (Why?)

L2.7 Retrieving System Information

The `System` class has a second static method:

```
String getProperty(String key)
```

This method takes a string that specifies the name of a property of the computer system on which the program is running, and it returns the value of that property. For example, this statement

```
output.println(System.getProperty("java.version"));
```

outputs the version of the JRE that is executing the statement. Several properties can be accessed through this method, and some of them are shown in Fig. 2.22. Create the program `Lab2_3` that outputs all properties whose names are shown in the figure.

Examine the output of your program. Does it correctly specify the name of the extension directory (where `type.jar` was stored)? It should also correctly indicate the name of your working directory (where the program was stored).

L2.8 Integrated Development Environments (*optional*)

The environment that was set up in this Lab requires that you switch back and forth between the editor and the console as you go through the edit-compile-run cycle. This is not a problem given that today's operating systems are multitasking: you can run the console without closing the editor, see both windows on the screen, and switch between them with

`java.version`	Java Runtime Environment version
`java.vendor`	Java Runtime Environment vendor
`java.vendor.url`	Java vendor URL
`java.home`	Java installation directory
`java.class.path`	Java class path
`java.ext.dirs`	Path of extension directory
`os.name`	Operating system name
`os.version`	Operating system version
`user.name`	User's account name
`user.home`	User's home directory
`user.dir`	User's current working directory

Figure 2.22 Some of the system properties that can be passed to the `getProperty` of the `java.lang.System` class