Lecture 2

Primitive Data Elements

Java has eight primitive data types: there names are boolean, char, byte, short, int, long, float, and double. For the time being we shall concentrate on three: boolean, int, and double.

The word primitive means that variables of these types can be declared in a java program without further instructions. If we want a variable m to be of integer type we make the declaration

```
int m;
```

within the code of a class. A variable x of type double is a number that has a decimal point; a variable p of boolean type is a variable that is either true or false. The declarations for them are

```
double x;
boolean w;
```

Declarations such as these merely define the type of the variable; they do not assign any value to it. If you wanted to assign the value 3.14159 to the variable y you would write

```
double y;
y = 3.14159;
```

The two commands could be combined on one line as

```
double y = 3.14159;
```

A variable of type int is assigned 32 bits in the computer's memory. This means that integer type variables can range from, roughly, $-2^{31}$ to $2^{31}$. To be more precise largest value a variable of type int can assume is 2,147,483,647; the smallest is -2,147,483,648. (reference: JDA, Java Developers Almanac, 2000, page 514) If an integer goes beyond these bounds arithmetic overflow occurs.

The value of a variable of type double is assigned 64 bits in memory. Numerically, the largest positive value a number of type double can assume is (about) $1.79769 \times 10^{308}$; the smallest positive value is $4.9 \times 10^{-324}$. (JDA, page 424)

An Example

This program illustrates some of the syntax employed in working with integers.

The user will input a positive integer n and the program will return n!; Recall, if n = 3 then 3! = 6; if n = 7, then 7! = 720.
The program will open by presenting a window that prompts the user to enter an integer:

Suppose the user types in a 7:

Then, after the OK button is clicked, the program responds with:

The program closes after the user clicks the OK button.

The skeleton of the program is

```java
import javax.swing.JOptionPane

public class Factorial
{
    public static void main(String args[])
    {
    }

    }// end main(String args[])

}// end class Factorial
```

The first line, `import javax.swing.JOptionPane`, says the program will use the "built-in" class `JOptionPane`, which is located in the file system `javax.swing`. This class is one of many that are part of the Java Swing system; it is used to produce the input and output screens above.
The next line, `public class Factorial`, says the program is called `Factorial`. Curly brackets

```
{
}
```

are used in java to mark the beginning and end of blocks of code. The double pair of slanted lines, `//`, is used to mark a comment. It is a good programming practice to introduce the brackets that begin and end a block of code and to comment where the block ends before writing any code. When our programs become more complex one can spend a great deal of time debugging, especially if the blocks are not marked.

Our class `Factorial` is going to have one main routine

```
public static void main(String args[])
```

This line is common to all java applications.

Although at this point there is no code defining what the class `Factorial` does it can be compiled:

```
Z:\11Pic20\Lecture_2\javac Factorial.java
Factorial.java:1: ":" expected.
import javax.swing.JOptionPane;
1 error
Z:\11Pic20\Lecture_2>
```

Compiling empty code after it has been outlined is a good practice because the compiler will indicate most simple syntax errors (misspellings, unbalanced brackets, etc) at compile time. In this case the underlying file has a syntax error; the required semi-colon at the end of the import statement is missing. So we go back and correct the error.

The complete program follows:
import javax.swing.JOptionPane;

public class Factorial
{
    public static void main(String args[])
    {
        String w;
        int n;

        w = JOptionPane.showInputDialog("enter a positive integer");
        n = Integer.parseInt(w);

        int product = 1;
        for (int i = 1; i <= n; i++) product = product*i;

        JOptionPane.showMessageDialog(null,
        "When n = "+n+", "+n+"! = "+product,
        "Results",
        JOptionPane.PLAIN_MESSAGE);

        System.exit(0);
    }
}

The first two lines of main(String args[]) block
        String w;
        int n;

illustrate two of java's types, the String and the integer, which
is denoted by int. The String type is not primitive, in that it
is built up of characters, which are primitive. Two variables w,
a String, and n, an integer are declared here.
The next line

JOptionPane.showInputDialog("enter a positive integer");

utilizes the built-in class JOptionPane. A "class" is a piece of software that is dedicated to one or more tasks. In this line it uses a method of the class, showInputDialog(), to produce the screen:

A method is sometimes described in other languages as a function or a procedure; java uses the word method.

The complete instruction

w = JOptionPane.showInputDialog("enter a positive integer");

has two purposes. First, it produces the screen shown above. Second, after the user types in an integer and clicks on the OK button, it captures the integer typed in. It is captured as the string w.

The string w has to be converted to an integer, n. This is done by means of the next line:

n = Integer.parseInt(w);

Integer is a java class that contains a method parseInt(w). The method takes the string w and converts it to an integer. We will discuss classes and their methods in great detail as we move along.

The line

int product = 1;

accomplishes two ends. First, it declares that product is a variable of type integer. Second, it initializes the value of the product to be 1. This needs to be done so that the for loop

for (int i = 1; i <= n; i++) product = product*i;

works correctly. Note that if a for loop has only one instruction in its defining block, like this one, then curly brackets , { and }, are not needed.

The lines
JOptionPane.showMessageDialog(
    null,
    "When n = " + n + ", " + n + "! = " + product,
    "Results",
    JOptionPane.PLAIN_MESSAGE
); can be read as follows. The class

    JOptionPane

will produce the output pane. This class has a method

    showMessageDialog( , , , )

for producing the output screen:

![output pane]

The showMessageDialog() method has 4 variables. These variables are: a variable called null, a string that will appear in the output, the title for the output Pane, and an instruction that says that the output is a message pane.

The string in

    "When n = " + n + ", " + n + "! = " + product

shows another feature of java: Strings, ordinary text included between double quote marks, can be combined with integers by means of + signs. The grand total will be interpreted and displayed as a string.

The last command

    System.exit(0);

closes the program when the user clicks the OK button in the output panel.

Arithmetic Overflow

Recall that a variable of type int, a 32 bit number, ranges approximately between $-2^{31}$ and $2^{31}$. We shall use the previous program to illustrate overflow.
To begin, we compute 12!

Since

\[ 12! = 479,001,600 < 2^{31} = 2,147,483,648 \]

we can feel confident that java has calculated 12! correctly.

Going on to the next case we get:

The calculation here is clearly wrong. By the previous result, 12! = 479,001,600 has two 0's as its last two digits (it's divisible by 100) so 13! would have to have at least two 0's at its' end.

The error was caused by arithmetic overflow: Since 12! is about 479 million, 13! = 13(12!) will be about 6 billion, which is much greater than \( 2^{32} = 2,147,483,648 \).

Exercises (from the textbook Deitel & Deitel)
Read Chapter 2. Then do
Pages 68– , problems
2.1, 2.2, 2.4, 2.7, 2.9, 2.10, 2.13, 2.14