Chapter 6. Transition to Java

Not all programming languages are created equal. Each is designed by its creator to achieve a particular purpose, which can range from highly focused languages designed for a particular activity or domain to more general-purpose languages designed to support a broad range of activities in a variety of domains. Processing is a more focused language. Its creators, Casey Reas and Ben Fry, built the Processing development environment to support the electronic arts and visual design communities by providing easy-to-use support for two and three-dimensional graphics, typesetting and animation. Initial results show that Processing has been successful in achieving this goal.[1] The first half of this text has explored these areas of the Processing language in some detail.

The second portion of this text begins to address the variety of activities and domains found in the broader field of computing. In particular, it will consider the development of larger applications that address domains outside the area of the electronic arts and visual design. Such applications, which might include dozens or even hundreds of classes and require support for tasks other than graphics, typesetting and animation, stretch the limits of the Processing language and development environment beyond what they were designed to support.

To support these larger applications, this chapter turns to the Java programming language and its more powerful development environments. Java is a general-purpose language whose creators, Patrick Naughton, James Gosling, Mike Sheridan and others at Sun Microsystems, envisaged a language that would support a wide variety of activities in a cross-platform environment. To achieve these goals, they designed an extensible language based on a virtual-machine architecture. Java has been wildly successful in achieving this goal.[2] It has been ported to a wide variety of computing platforms and has attracted the development of numerous, enterprise-level development environments. Indeed, the Processing language itself is built on Java, which makes our "transition" from Processing to Java less like a transition and more like a natural next step.

This chapter introduces the Java programming language.

6.1. Example: A Tip Calculator

To this point in the text, our examples have been primarily graphical in nature. This is in keeping with the nature of Processing, which casts every problem as a graphical problem. If, on the other hand, we're addressing a non-graphical problem, say the desire to have a simple calculator that takes a restaurant bill and automatically computes an appropriate tip, we're likely to find Processing's insistence on producing graphical output more of a nuisance than a benefit. All we really need in a context like this is a program

[1] For a selection of Processing examples, see http://www.openprocessing.org/.
[2] For a section of Java examples, see http://www.java.com/en/java_in_action/.

that allows the user to enter a floating point number representing the amount of the bill and then computes and prints an appropriate tip. No graphics are required.

The interaction with this program might be something like this:

```
Please enter the cost of the meal: $20.00
Appropriate tip: $3.00   (based on a 15% tip rate)
```

Here, the program output is shown in black and the user input is shown in green. One could imagine a nicer interface using buttons and numeric fields, but the sort of two-dimensional graphics in which Processing specializes wouldn't be all that useful here.

This section introduces the mechanisms provided by the Java programming environment that can be used to implement this behavior. Additional tools for graphical user interfaces are addressed in a later chapter.

6.2. Java Basics

Processing is based on Java and runs on the Java virtual machine, so the syntax of Processing statements is the same as the syntax of Java statements. However, the Processing development environment automatically performs a number of important programming tasks that Java programmers must perform manually. This makes Processing programs easier to write but also tends to favor a narrower range of graphically-oriented goals. By contrast, Java programs are harder to write but can be designed to achieve a broader range of goals. By the end of this chapter, we will see that Processing programs fit nicely into the broader context of Java programs, but we must start by covering some basic elements of Java programs.

6.2.1. Writing, Compiling and Running a Simple Java Program

In Java, every application is implemented as a class with a `main()` method. The following Java program is a maximally simple, do-nothing program that runs but accepts no input and produces no output.[3]

```
public class DoNothing {

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

}
```

The syntax of the **`main()`** method is relatively familiar given our experience with writing methods in Processing, but there some new programming constructs in this program:

- A Java program is implemented as a class using the pattern:
  `public class` ***`ClassName`*** **`{ }`**.

[3] While the Processing language is supported by only one Integrated Development Environment (IDE), see the discussion in Chapter 1, the Java language is supported by a wide variety of IDEs, including BlueJ, Eclipse, JBuilder and NetBeans. This text will not pre-suppose any particular IDE; the lecture and lab materials present particular IDEs.

We will discuss classes, their definition and their use in Chapter 9. For now, simply note that all Java programs must be declared in a new class, as shown here, and stored in a file whose name is the same as the class name, with the **`.java`** extension.

- A Java program is implemented as a class with a single **`main()`** method. This "main" method must be called **`main`**, be declared as **`public static`**, return **`void`**, and declare an array of Strings as a parameter.[4] When we run this class as an application, the Java environment automatically calls this method. This approach is similar to the way in which Processing automatically calls the **`setup()`** and **`draw()`** methods when we run a Processing program.
- This Java program specifies that the class and the method should be **`public`**. We can use **`public`**/**`private`**/**`protected`** access level modifiers in Processing as well, but only in the context of class data and methods. In this example, the **`public`** modifier is required for the **`main()`** method and we make it a practice of always using **`public`** access for classes as well.
- The **`main()`** method is declared to be **`static`**, which specifies that Java should associate the **`main()`** method with the **`DoNothing`** class rather than with individual objects of type **`DoNothing`**. We discuss this concept in more detail later in the text. For now, make it a practice of declaring the **`main()`** method as static. As we see below, we'll also declare constants to be static.

The Processing development environment automatically adds this supporting code to your Processing application in order to run it as a Java application. You can take this for granted with working with Processing, but when you move to other more traditional programming applications, you must turn to Java and build this code yourself.

We now discuss these issues in further detail and take the opportunity to extend this do-nothing application into a do-something application.

6.3. Building Java Console Applications

Processing programs are exclusively graphical in nature. Even the following Processing program, which prints a message on the text output panel, brings up an empty graphical output panel.

```
println("simple text output");
```

This reflects Processing's pointed focus on graphical applications.

Java programs, on the other hand, may or may not be graphical. The simplest Java programs are textual and conduct their user interaction by prompting users with text messages on a text-only command screen and reading user input from that same screen. The command screen is commonly called a *console* and text-oriented applications are commonly called *console applications*.

[4] The `args` parameter receives the arguments passed to the program from the command-line. More details on this feature of Java can be found here: http://java.sun.com/docs/books/tutorial/essential/environment/cmdLineArgs.html.

For example, the following Java program implements the same behavior as the Processing-based simple text output program shown above.

Java Program:

```
public class SimpleOutput {

	public static void main(String[] args) {
		System.out.println("simple text output");
	}

}
```

Sample Output:

```
simple text output
```

This Java version of the program requires a bit more code to implement the same textual output and doesn't produce a graphical output screen. More work for less functionality – this doesn't feel like a step in the right direction.

However, the Java environment provides one feature that we've been wishing for throughout most the previous chapters, the ability to get textual input from the user. Consider the following program, which reads the user's name and prints out a personalized message.

Java Program:

```
import java.util.Scanner;

public class SimpleInput {

	public static void main(String[] args) {
		System.out.print("Please enter your name: ");
		Scanner keyboard = new Scanner(System.in);
		String name = keyboard.nextLine();
		System.out.println("Let it be known that " + name + " was here.");
	}

}
```

Sample Interaction:

```
Please enter your name: Kilroy
Let it be known that Kilroy was here.
```

This sample interaction shows the user input in `green`; the computer output in `black`. When the user runs the program, Java prints the initial prompt ("`Please enter your name: `") and then waits for the user to enter a string of characters. The user must then enter their name and press the "Enter" key. The program then reads the remainder of the line and then formats and prints an output message.

Note that while this example program seems rather simple, it is doing one thing that Processing doesn't easily support – it reads a string typed in by the user. Processing supports text output to its text output panel, but to implement text input in a Processing program, the program would need to use

`keyPressed()` to listen for key strokes from the user and convert the sequence of characters representing a name into a string, one character at a time. While this is possible, it is not that easy, and the task would be more difficult for numeric input in which the program would need to convert sequences of numerals into numbers. For example, it would have to convert the five-character sequence: `'4' '8' '.' '2' '4'`, into the decimal number `48.24`. As we will see in the next section, Java's `Scanner` class provides a pre-defined algorithm for this parsing problem.

This section now talks about the key features of this program.

6.3.1. Importing and Using Java Packages

In Processing, we tended to simply use pre-defined features such as **`print()`** without much fanfare; they were simply there for us to use.[5] In Java, we must explicitly load pre-defined features using the **`import`** command. In the example, the following command is used to load the **`Scanner`** class library.

```
import java.util.Scanner;
```

This command is placed at the beginning of the class file and must specify the fully qualified name of the Scanner Class, which includes the **`java.util`** package designation. Without this import command, Java would not have access to the definitions for the input parsing methods, e.g., **`nextDouble()`**.

The Scanner class defines a number of useful input methods, including:

- **`nextDouble()`** reads double values;
- **`nextInt()`** reads integer values;
- **`next()`** reads the next word;

Note that the **`Scanner`** class reads through the user's input one character at a time, scanning for appropriately structured sets of numerals, decimal points, letters and spaces that make up numbers and words. This process is called *parsing*, A sequence of numerals like "123" is interpreted as an integer; "abracadabra" as a word; "3.14159" as a floating point number. The scanner knows when a number or word starts and stops by looking for *whitespace* characters, that is spaces (" "), tabs and new lines. When reading a double, integer or word, the scanner reads the item in question and then consumes (i.e., skips over) the white space character that follows it.

Reading a string that includes spaces, like "to be or not to be," requires the use of another method:

- **`nextLine()`** reads the remainder of the line as a `String`.

This method preserves the spaces and tabs in the string, consuming only the new-line at the very end.

A full discussion of the Scanner class and its methods can be found elsewhere.[6]

---

[5] Processing does support imported libraries, which we used to load and play audio files using the Minim library, but most of the basic graphical features are pre-loaded.

[6] A full discussion of the `Scanner` can be found here: http://java.sun.com/docs/books/tutorial/essential/io/scanning.html.

6.3.2. Working with Java Objects

The Java programming language is more overtly object-oriented than Processing. The first clear example of this fact is that even the do-nothing program shown above is implemented as a class. The same is true with the **`SimpleOutput`** and **`SimpleInput`**. When we ask Java to run these classes, it looks for a **`main()`** method with the appropriate signature, i.e., **`public static void main(String[])`**, and invokes that method.

A second example of this object-oriented focus is the commands used for text output and input. Consider the following print command:

```
System.out.print("Please enter your name: ");
```

This command is the same as it is in Processing except that we've added the **`System.out`** reference as a qualifier. All calls to methods in Java must be fully qualified with the object or class that defines them. In this case, the **`print()`** method is defined by the **`out`** object, which is a public data item defined in the **`System`** class that always references the standard output stream to the console. The same use applies to the **`println()`** statement at the end of the program. We use the dot notation discussed in Chapter 3 to specify that these methods are defined for the **`System.out`** object.

The same rule is applied to the commands in the example that read user input from the console.

```
Scanner keyboard = new Scanner(System.in);
String name = keyboard.nextLine();
```

The program uses the **`Scanner`** class to read user input. It declares a **`keyboard`** object of type **`Scanner`**, passing it the system-defined input stream, and calls this new **`Scanner`** object's **`nextLine()`** method to read the remainder of the current line typed by the user. This call is done, again, using dot notation. Finally, the Scanner constructor receives as an argument **`System.in`**, an object that references the standard keyboard input stream from the System class using dot notation.

The general pattern for using the dot notation to reference objects and methods is as follows:

Member Reference Pattern

objectIdentifier.*instanceMemberIdentifier*
OR
ClassIdentifier.*staticMemberIdentifier*

- *objectIdentifier* is a reference to a particular object that defines the instance member;
- *instanceMemberIdentifier* is the name of the instance data or method member being referenced;
- *ClassIdentifier* is the name of the class that defines the static member;
- *staticMemberIdentifier* is the name of the static data or method member being referenced.

Printing strings, as shown here, is the same in both Processing and Java. Processing, however, doesn't provide support for formatting numbers. Java does. For example, the following program reads a number and computes the interest earned based on a 6% interest rate:

Java Program:

```
import java.util.Scanner;

public class ComputeInterest {

	public static double RATE = 0.06;

	public static void main(String[] args) {
		System.out.print("Please enter your account balance: ");
		Scanner keyboard = new Scanner(System.in);
		double balance = keyboard.nextDouble();
		System.out.printf("\tYour interest: %.2f", balance * RATE);
	}
}
```

Sample Interaction:

```
Please enter your account balance: $1000.00
	Your interest: $60.00
```

This program declares and initializes **RATE**, a constant to represent the interest rate. As noted in the introduction, constants like this are declared to be **public** and **static** as done with the **main()** method. We cover this issue in detail in Chapter 9.

This program uses the **printf()** method, which is also defined by **System.out**. This is a print statement, similar to **print()** and **println()**, that supports user-specified formatting. The first argument to **printf()** is a String to be printed, called the *format-string*. In addition to normal characters, the format-string can contain a set of formatting specifications known as *placeholders* where each placeholder must be matched by an additional argument following the format-string. The **printf()** method replaces each placeholder with the value of corresponding argument formatted a specified by the placeholder. The following illustration demonstrates how the **printf()** method works in the given example:

Java Program:

```
System.out.printf("\tYour interest: $%.2f", balance * RATE);
```

Sample Output: Your interest: $60.00

Here, the placeholder **%.2f** is replaced in the output string by the value of the expression **balance * RATE**, formatted as a decimal number with at most two decimal digits.[7]

[7] The Java API reference documentation provides more detail on the use of the Scanner class, see http://java.sun.com/j2se/1.5.0/docs/api/java/util/Scanner.html.

6.3.3. Adding methods to Java Applications

Notice also that the **RATE** constant definition is placed in the **ComputeInterest** class definition, that is, inside the class definition block. This makes the constant local to the **ComputeInterest** class, accessible to all the methods that may be defined for **ComputeInterest**, which, at this point, is only the **main()** method.

If we chose to implement a method to compute the interest, this method would also be defined in the **ComputeInterest** class as shown here.

Java Program:

```
import java.util.Scanner;

public class ComputeInterest {

	public static final double RATE = 0.06;

	public static double computeInterest(double balance) {
		return balance * RATE;
	}

	public static void main(String[] args) {
		System.out.print("Please enter your account balance: $");
		Scanner keyboard = new Scanner(System.in);
		double balance = keyboard.nextDouble();
		System.out.printf("\tInterest: $%.2f (using a %2.1f%% rate)",
				computeInterest(balance), RATE * 100);
	}

}
```

Sample Interaction:

```
Please enter your account balance: $100
	Interest: $6.00 (using a 6.0% interest rate)
```

In this code, the **RATE** constant as well as the **computeInterest()** and **main()** methods are declared in the **ComputeInterest** class definition block as **public** and **static**. **RATE** is accessible to both methods and is, indeed, used by both for their own purposes, **computeInterest()** for computation and **main()** for display.

6.4. Revisiting the Example

Returning to the vision of a tip calculation program, we would like to implement the following algorithm:

Given:

- A specified tip rate (e.g., 10% or 15%)

Algorithm:

1. Print a prompt for the total restaurant bill on the screen.
2. Read a double value from the screen.
3. Print the suggested tip on the screen.

This is an interactive console program that can be implemented using the mechanisms discussed in this chapter as shown here.

Java Program:

```
import java.util.Scanner;

/**
 * TipCalculator computes a tip based on a total restaurant bill and a fixed tip
 * rate.
 *
 * @author kvlinden
 * @version Fall, 2009
 */
public class TipCalculator {

	private static final double TIP_RATE = 0.15; // The hard-coded tip rate

	/**
	 * This method drives the console-based interface for the application.
	 *
	 * @param args
	 *            ignored
	 */
	public static void main(String[] args) {
		System.out.print("Please enter the total bill: $");
		Scanner keyboard = new Scanner(System.in);
		double bill = keyboard.nextDouble();
		System.out.printf("\tSuggested tip: $%.2f", bill * TIP_RATE);
	}

}
```

Sample Output:

```
Please enter the total bill: $48.24
	Suggested tip: $7.24
```

Note that the format and content of the documentation in this program is very similar to the documentation used in Processing applications. In Java, the main application documentation is generally placed just above the class definition, after the package and import commands.