Programming via Java-Programming overview

In this book, we examine the basics of computer programming. We will study this using one particular language, Java. But the point is not to master Java (which would takes several courses). Instead, we regard the language as a tool for learning the fundamentals of expressing systematic approaches to solving problems. By the end, you should have acquired the skill of expressing complex procedures using Java.

This is not the only Java textbook that is available on-line without requiring payment. In the spirit of encouraging free discourse and usage of these on-line resources, below is a list of the ones of which I am aware. (They vary in how permissive they are with permitting copying and modification of materials.)

1.1. About Java (optional)

In the early 1990's, a computer company called Sun Microsystems began a project to develop a platform for *embedded systems* — that is, they wanted to build software for devices that have a special-purpose computer inside them, like a microwave, a telephone, or a car. The people assigned to the project began by asking what sort of language they would want to use for such a system. They decided that none of the alternatives were suitable, and so they developed Java. In the wake of the effort to develop a new language, their embedded systems mission fell by the wayside.

When Sun released Java in 1996, the company had enough marketing sense to throw in features to support the hottest technology of the day, the World Wide Web, via the concept of an *applet*. That idea was only a moderate success — except that the association with the Web gave the language all the hype that it needed to become widely known.

In developing Java, its designers drew on a long history of programming languages. Historically, Java derives primarily from two languages: C and Smalltalk. Designed in the 1970's for developing an operating system called UNIX, C is a small, efficient language. But it provides few *abstractions* — a C programmer thinks in terms of how the computer actually computes, which makes it difficult to build large programs to do something useful. It became very popular in the industrial world, however, due largely to the increasing popularity of UNIX. In fact, the primary reason Java builds on C is so that the vast body of C programmers would accept Java.

Smalltalk, also designed in the 1970's for an operating system, is an elegant, inefficient language. In contrast to C, Smalltalk provides many abstractions; a Smalltalk programmer rarely considers or even understands how the program works within the computer. Although Smalltalk never achieved the popularity of C, it was popular enough that the value of its fundamental abstraction — the *object* — began to be appreciated by many programmers. (The concept of objects originated 10 years earlier in a language called Simula, but Smalltalk popularized the concept.)

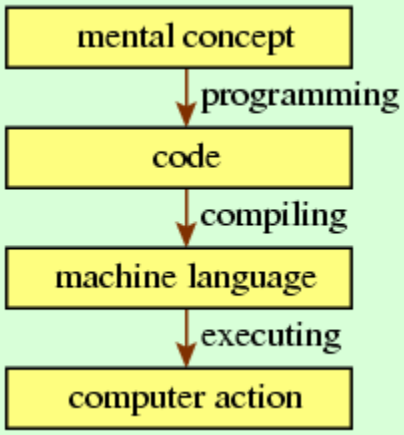
Java's designers sought to combine the abstractions of Smalltalk with the efficiency of C. Their design has proven a sound response to a long-established need, which has enabled it to gain a foothold in industrial programming. It's difficult to quantify how widely used a language is, since a large fraction of software is developed by individual companies for their own internal use. But Java is likely among the top five most-used languages in large-scale systems today.

It is easier to quantify Java's success in education. Most university computer science curricula now use Java as their primary language. This future workforce trained in Java virtually guarantees its endurance in the workplace.

1.2. The programming process

The pipeline from idea to action consists of three phases: *programming*, *compiling*, and *executing*. Figure 1.1 illustrates the process.

Figure 1.1: The programming process.



In programming, you as the programmer translate your mental concept of how the computer should behave to corresponding code written in a programming language such as

Java, C, or Smalltalk. A programming language is a compromise between what humans find natural for expressing procedure and what computers can interpret.

Computers, as built, cannot actually understand programming languages; they are built to understand a much more primitive language called machine language. (Different types of machines have different machine languages — the machine language for a PC is totally different from the machine language for a typical gaming console.) In compiling, the computer executes software to translate the programmer-written code to machine language. This software is called a compiler, and it is the primary programming tool for the compile phase.

In the final phase, the computer executes the machine language that the compiler produced. At this point the machine finally does the job that the programmer originally conceived... at least if all the phases proceed flawlessly.

During these phases, errors crop up due to programmer errors. For all these errors, the solution is for the programmer to discover where the code is wrong, to fix the code, and to repeat the compile and execution phases.

- **Programming:** A logic error arises when the programmer has written code that compiles and executes normally, but the machine's behavior does not correspond with the original concept. An example of a logic error would be if a program intends to identify when the user clicks on a shape in the window, but the program fails to include clicks on the shape's outermost boundary.
- **Compiling:** A compile-time error is an error that prevents the compiler from interpreting the program. If the code contains a mistyped name, for example, then this causes a compile-time error. The result of a compile-time error is that the compiler refuses to compile the program, instead issuing a description of what is wrong with the program. This is the easiest type of error to fix, since the compiler usually points directly to the problem.
- **Executing:** A run-time error occurs during execution; one example of a common run-time error is when the machine code instructs the computer to divide a number by zero. Often such errors *crash* the program; that is, execution stops abruptly.

1.3. Libraries used by this book

This book will use two libraries that extend Java specifically for educational purposes. We could learn Java without these extensions, but we wouldn't be able to use graphics well, and from the beginning you'd have to deal with several concepts that you wouldn't really understand until near the book's end.

- The first of the libraries, named `turtles`, is a very simple library involving an animated turtle drawing lines on the window. It may sound a bit childish, but it's a good way to get acquainted with the system. This library was developed by the author especially for this book. We'll use this library at the beginning of this book, up until Chapter 5.
- The second is a much bigger library developed by a committee of several computer science educators associated with an organization called ACM SIGCSE.Chapter 6 will introduce this library, and we'll continue with it for most of this book.

To execute the programs in this book, and to write your own using these extensions, you should download the libraries. They are packaged together into one file, called*turtles.jar*. On most Web browsers, you can download the file by right-clicking (or, on MacOS, control-clicking) the link turtles.jar and choosing Save Link As... or something similar.

After downloading the file, you'll need to tell your Java compiler where it can find the libraries. This varies dramatically from compiler to compiler; your instructor should provide instructions.

1.4. A simple program

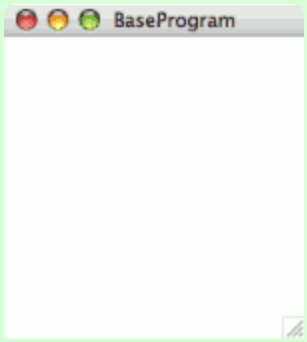
To begin our study of Java, we'll look at the program of Figure 1.2, which we'll use as the foundation for building programs in the next several chapters. [To facilitate reading, this book displays programs with line numbers and varying colors and typefaces (boldface, italics). These are not part of the actual program; the program contains simply the sequence of letters.]

Figure 1.2: The `BaseProgram` program.

```
1 import turtles.*;
2
3 public class BaseProgram extends TurtleProgram {
4     public void run() {
5         // This is a simple baseline program.
6     }
7 }
```

When executed on a computer, this program brings up an empty window on the screen, as in Figure 1.3. It's not too impressive, but we have to start somewhere.

Figure 1.3: Running `BaseProgram`.



For now, you don't really need to understand Figure 1.2. We will concentrate wholly on what can go in place of line 5. But when you write a program, you will need all of the other lines, too, and so you can simply copy the rest of lines, placing the name of the file on line 3.

You won't need to understand those other lines until much later. That's scant consolation to the curious, however — and we don't want to squelch curiosity. So let's do a quick overview to see what this program says.

- **Line 1:** This tells the Java compiler that this program is going to use pre-written code defined somewhere else — specifically, in the `turtles` library.
- **Line 2:** Java ignores blank lines. They make the program a little easier to read for humans. The program works just as well without line 2, but it would be more confusing for a programmer to understand.
- **Line 3:** This tells the Java compiler that we are defining a program named `BaseProgram`. The program goes between the left brace at the end of line 3 and the corresponding right brace in line 7.
- **Line 4:** This tells the Java compiler that we are defining what the program is to do when asked to run. Everything between the left brace on line 4 and the corresponding right brace on line 6 contains what the program should do when it executes.
- **Line 5:** This is a comment — that is, it is part of the program that the computer simply ignores. This is useful for describing what the program is trying to do to a human trying to understand it. You can indicate a comment in Java using a double slash ('`/`'); Java will ignore everything on the line starting from there.

1.5. Programming style

When you write a program, you're really writing for two audiences. The obvious audience is the computer that will be executing the program. But just as important is the human

audience — somebody who needs to evaluate your program's correctness or perhaps even to modify it to incorporate new features. You should think of this person as coming in with an understanding of Java and what the program accomplishes, but who has no real idea of how your program works. That person may be your instructor, your boss, your coworker, or it may even be you after you've been doing other things for a few months and then decide to go back to change that program you were working on before.

Such people will need help to understand easily how the program is structured to do what it does. Thus in writing a program, we desire *readability* in addition to *correctness*. Java includes three major features in order to facilitate readability: *white space*, *naming*, and *comments*.

White space

The term *white space* refers to characters of a file that you can't see — spaces, tabs, and line breaks. In Java, white space does not matter. In fact, Java regards the following program as being identical to that of Figure 1.2.

```
import turtles.*;public class BaseProgram extends TurtleProgram
{public void run(){}}
```

While humans may not care, however, white space can be very useful to humans in indicating the structure of a program, particularly if used systematically.

In this book, we'll faithfully follow one particular approach for using white space.

- We put one concept on each line. Typically, the concept will be a single statement directing the computer to do something. (Our first program, though, contains no statements.)
- We insert a blank line between largely independent pieces, akin to how authors insert paragraph breaks in English text.
- We'll indent everything within a set of braces four spaces. The `BaseProgram` program demonstrates this: Lines 4 to 6 are all within the set of braces begun at the end of line 3, so they're all indented four spaces. Moreover, line 5 is within the set of braces begun at the end of line 4, so it's indented an additional four spaces.

These are conventions followed by a large group of Java programmers, based on years of experience writing millions of lines of code. You should get into the habit of using these conventions too. (If you have programmed a lot before and already use another widely used convention, that's fine too. But don't go off and develop your

own — it will confuse others and cause more headaches than it's worth in the long run.)

Naming

The second element of good programming style is using good, descriptive names. When you program, you have lots of opportunities to assign names to things. Choosing representative names is worth some consideration. For example, in the `BaseProgram` program, I arbitrarily chose the name *BaseProgram*. I could have chosen virtually anything.

```
public class BenAffleck extends TurtleProgram {
```

Even though it's legal, don't choose silly names like *BenAffleck*. Instead, choose names that communicate something about what the program is doing with that name. The name *BaseProgram* is much better: It communicates both that this is our first program, but also that this program represents the basis from which you can build later programs.

Source: http://www.toves.org/books/java/ch01-overview/index.html