Chapter 2 Programming the Simplest Java Programs

Introduction

This chapter introduces the structure of Java programs through several examples. It also presents an important problem-solving technique, namely learning to define problems in such a way that you can come to grips with their essentials instead of becoming hopelessly mired with nonessential issues. In this spirit the examples are presented without complete explanations of the intricacies of every component; a thorough presentation will be deferred until Chapter 5. This is done for two reasons. First, it is more interesting to write programs and see them work than to memorize long lists of details. Second, it is much easier to understand a programming language after you have had some experience working with it.

Learning to Program

Once, a traveling saleswoman was driving down a country road and spied a pair of coveralled legs protruding, unmoving from under a gigantic combine. Concerned for the well-being of this prostrate farmer, she stopped her car, hopped the fence, walked up to him, and asked, “Are you okay?” “Yep, just trying to fix this combine,” he replied. The saleswoman studied the bewildering complexity of gears, levers, hydraulics and wires for a moment, then asked, “How did you learn to fix such a complicated thing?” “Don't know a thing about fixing 'em,” he replied. “Then how on earth do you expect to be able to fix it?” “Well...” drawled the farmer, “...I figure a person made it.”

A programming environment is a little like that combine. It is composed of many interacting parts and at first it certainly looks like you might never figure it out. But, once you learn how to, programming is easy. The computer always does exactly what you tell it to! Programming is also fun, once you get the hang of it. Learning to program may seem a daunting task. To the uninitiated, the complexity may appear overwhelming, incomprehensible. Many interacting parts must be in the proper relationship before even the simplest program will work.

Like that combine, the Java language is large and complex, but, again like the combine, it is very useful once you know how to use it. The old C programming language evolved into the object based language C++ and then into Java. As a result, Java has some complicated syntax. If a novice programmer were required to understand everything in a first program, it might take weeks or months before that program could be written. But, this book uses a spiral approach. First the essentials of Java are introduced by way of examples. Those concepts will then be revisited in more detail in Chapter 5; this will provide the understanding to accomplish many programming projects.

Learning to program in Java, like fixing a combine, is possible. Learning to control the information processing machine we call a computer is initially a lot of work, but after surmounting that initial obstacle it can also be exciting and rewarding.
Problem Solving

“There are only two kinds of proofs in mathematics -- trivial proofs and impossible proofs. The difference is that the trivial ones you already understand”.

Freshman Calculus Prof.

The author of this text is an experienced programmer. He has written large, complex software in numerous programming languages. But, when he first uses a new programming language he writes a tiny program that displays the single word, “greetings”, on the screen. Why is this? The answer is simple, but difficult for some to learn. Do you know (without looking ahead) what it is? If you do, you already know one of the most important lessons of problem solving.

Definition of Problem Solving

Life is full of problems. You are already an expert problem solver, otherwise you would not be where you are now, and you would not be reading this book. Problem solving (as used in this context) is different from ordinary functioning or coping. Every day you find your way home to sleep. Sometimes your car breaks, or your bike is stolen, or you’ve sprained your ankle, or there is a traffic jam, or road construction, or a train blocks the way. You deal with those problems and get home anyway, but that’s mostly just coping. Problem solving is what you do when you are stuck; when you’re stopped and don’t know what to do next. That’s never a pleasant position to find yourself in, but if you develop your problem solving skills, it is a lot more comfortable (and less common!).

Details

Here’s a fact that a novice programmer might not know: When you first learn to program you get stuck frequently. Why? Because programming requires familiarity with a large number of details. Worse, when you are starting out, since you don’t really understand what you are doing, that mass of details doesn’t really mean anything, so there’s no way to remember them all. Later, as you become familiar with the process, and develop good programming habits, you’ll deal with many of those details automatically without having to think about them. But initially, even though the individual details are small and not especially interesting, every one of them must be correct for your program to work.

It’s a little like when you move to a new city. At first you only know where a few places are, and getting anywhere is a mystery and takes concentration. Then you learn a few places and routes, but anytime you get off them, you are lost. In time, it becomes familiar and you don’t even have to think about how to get places. But at first it’s all new and confusing. Programming too. But it does improve.

Bugs

Errors that occur when a program runs are called bugs. There’s a story that one early computer didn’t work for days until someone discovered a dead moth between a contact. The story may be apocryphal, but the term is here to stay. An accomplished programmer has much experience solv-
ing the problems that arise in the course of programming. In other words, part of learning to pro-
gram is learning to deal with bugs; both finding and fixing them (and the former is sometimes
harder than the latter!).

A Tool Kit for Solving Problems

This text will include a number of problem solving techniques. Some of these will be specific to
programming in Java, but the majority will be more generic. Think of these techniques as tools.
If you learn these techniques, then when you confront problems, both programming and other-
wise, you may feel better prepared. As if you’ve acquired a toolbox of useful problem solving
methods. Instead of discovering yourself stuck and feeling like you don’t know what to do, you
may find yourself saying, “Huh! Stuck!”, and then immediately start to consider which of the
various techniques in your toolbox is most likely to be helpful.

A Problem Solving Exercise

Almost everyone has been exposed to the Pythagorean Theorem: in a right triangle, the square of
the hypotenuse is equal to the sum of the squares of the other two sides. Consider Figure 2.1 as a
circle centered at the origin, together with a rectangle inscribed in the upper right quadrant. The
radius of the circle is \( r \). The height of the rectangle is \( y \). The question is, what is \( h \)? If you knew
the width of the rectangle you could use the Pythagorean Theorem, but all you know is the ra-
dius, \( r \). Try to figure out \( h \); but don’t spend more than a few minutes on it. The answer is later on

![Figure 2.1 The radius, \( r \), and the height of the rectangle, \( y \), are known. What is \( h \)?](image)

in the chapter.
The Two Types of Java Programs: Applications and Applets

Applets

Applets are programs that can run in Web browsers. They are commonly used to add functionality to Web pages. When you use the Web, if you pay attention, you will sometimes see the message “Starting Java” in the lower left corner of the screen. It is very convenient to be able to execute Java Applets from Web pages; indeed, Applets were designed for this purpose. But, there is a problem. When you go to someone’s page and an Applet starts running, it is running on your machine. If there were no safeguards, this program, that you might not even know is running, could be damaging your computing system, or stealing all the information from your disk. To avoid this possibility (and thus to convince people to allow Applets to start up in browsers at all!) Java has a Security Manager, that limits what Applets are allowed to do. The most important things that Applets cannot do are reading and writing files on your local computer, and opening sockets to communicate with other machines.

Applications

Applications run independently; they do not depend on a browser (although they do require a Java Virtual Machine to be installed). They cannot be embedded in a Web page. But, they are not prohibited from file I/O, or opening Sockets; they are full-fledged programs.

Components of a Java Program

What are Java programs made of? Abstractly, a program is a series of symbols that conforms to the grammatical rules (also called syntax) of a particular programming language.

One of the big advantages of object oriented programming is that code is reusable. Packages allow you to easily import classes that other people have written into your program. These classes include input/output mechanisms (to move information into and out of programs), mathematical functions, screen control and many other things.

Class declarations include both memory declarations (which declare variables to store information) and method declarations. Method declarations explicate how members of a class respond to the messages of the same name. A useful metaphor is to think of the objects in a program as actors, and methods are the scripts the actors will read when they get cues (messages). The main method is the main script for an Application.

A Template for Every Java Main Method

Every Java Application must have a method named “main”. This is the method that will be executed when the program runs. The main() method must have the structure shown in Listing 2.1.
When a program runs, it executes each line in the `main()` method from the top down. The general declarations may create objects and the statements may send them messages. After the last statement is executed and the closing `}` is reached, the `main()` method, and thus the Application, is terminated.

**Formatting/Conventions**

In this text Java class names, variables and methods will be printed in *Courier* font to make them visually distinctive. Like `main()` in the previous paragraph. Methods will be followed by `()`’s, and class names will start with capital letters.

**A Template for a class to contain a Main Method**

Methods cannot exist on their own, they must be contained in a class. Like this:

```java
    class Foo {
        public static void main (String [] args) {
            General Declarations
            Statements
        } // end of the main method
    } // end of the class named Foo
```

This creates a class, named `Foo`, with one method, named `main()`. Notice that lines 1 and 6 bracket the code from Listing 2.1 with `{}`’s. These curly brackets (also called curles) are what Java uses to delimit sections of code. If there are different numbers of open curles and close curles, the Java compiler will not be able to make any sense of the code. Pay attention to the curles!!

Here is the same class with the curles emphasized:

```java
    class Foo {
        public static void main (String [] args) {
            General Declarations
            Statements
        } // end of the main method
    } // end of the class named Foo
```

Notice that the great big ones (on lines 1 & 6) delimit the class Foo, whereas the smaller ones (on lines 2 & 5) delimit the main method.

The curly brackets can blend in; *you* must pay attention to them (remember, *you* are the intelligence/adaptive system in the computer/human system). The ability to pay attention to the
right things is critical to programming (and many other things -- selective attention is an important part of expertise), and curly brackets are what the Java compiler is using to determine where things begin and end. Keep your eye on them!

Examples

This section has three examples: 1) the simplest possible Application, a robot greeter; 2) a slightly elaborated version of that, a personalized greeter; and, finally, 3) a very simple Applet.

The First Example of a Class -- A Robot Greeter

Imagine you have decided to build a robot. Before going to all the work and expense of buying and assembling the motors and gears and whatnot, you wisely choose to simulate a prototype first. You decide to write a Greeter class which (recalling the problem solving advice above) initially does nothing but say hello. Listing 2.2 shows this code.

Listing 2.2: A Program to Output “Greetings!”

```java
public class Greeter {
    public static void main (String [] args) {
        System.out.println("Greetings!");
    }
}
```

This program has only one class, named Greeter, with only the main() method whose heading reads: public static void main(String [] args)

That's a mouthful. Think of it as an incantation for now; like “abracadabra!”! You can copy and paste it when you need it (maybe even off this PDF if you have a modern PDF reader!).

Like any language, the Java programming language contains a number of idioms, sequences of symbols whose meaning cannot be derived from the individual symbols, but which must be learned as a whole, by rote.

This program is not particularly useful. The only reason for it is to help you learn to define classes in a simple setting where you can easily focus your attention on the essentials (This is the reason, if you hadn’t guessed, why experienced programmers write a greetings program first!).

Problem Solving Technique: Create a prototype before attempting any difficult project.

If there are too many details, too many parts, too many things to keep in mind, sometimes making the problem smaller or simpler will allow you to solve it and then adapt that solution to the larger problem.

Here is the explanation of code Listing 2.2 (reproduced here).
Listing 2.2 (reprise): The Greeter class declaration.

1    public class Greeter {
2        public static void main (String [] args) {
3            System.out.println("Greetings!");
4        }
5    }

**Line 1:** Every class declaration begins with the word `class`. After `class` comes an identifier, the name of the class being declared. **Identifiers** must begin with a letter and be composed only of letters, numbers and underscores. The name of the class is `Greeter`.

**Lines 1-4:** Next comes a pair of curly brackets containing a list of the memory and methods of this class. Here there is no memory and only one method, named `main()`. Every Application you write must have a `main()` method, and that method must have the exact heading on line 2.

**Line 2:** `public static void main (String args [])` is the heading of the main method. For now consider this an idiom or an incantation.

**Line 3:** The “body”, or script, of the `main()` method follows that heading. Like the script of every method, it is enclosed in `{}`s. The main script for a Greeter consists of the single line: `System.out.println("Greetings!");` This is an idiom meaning display the literal string of characters “Greetings!” on the screen.

**Running Your Program**

Run this program before proceeding. I will demonstrate during class, and you can try it yourself.

If you have never programmed, once you have compiled and run this tiny program, you will have taken a big first step. After a bit of practice it should only take a minute to do. Unfortunately there are many little things that you must learn, so the first time could take much longer if you had not seen the demonstration.

Speaking of solving puzzles, here’s a clue; the two diagonals of a rectangle are equal. Now do you know what \( h \) is? Sorry about the Pythagorean misdirection; the point was that how you look at a problem may determine how difficult it is to solve. If you were determined to calculate the width of the rectangle, you were doomed from the start. This is an instance of the following problem solving technique.

**Problem Solving Technique:** *Adopt a different perspective.*

*How difficult a problem is depends on how you look at it. A problem that seems impossible from one angle is sometimes trivial from another.*
Mechanics: Typing and Running Your Code

Whatever system you are using there are three steps to writing and running a program:

1) Creating source files: the Java code in the listings is called source code, when it is stored in a file that is called a source file. Netbeans writes the shell of your classes for you. Since the source code declares a public class called Greeter, Java insists that it be stored in a file called Greeter.java. Also, you must remember that capitalization matters! If you type PUBLIC static void main, it will object. If you type (string[] args) it will not know what string is; it only understands String. You must learn to notice the difference between String and string; it’s easy if you are thinking about the capitalization of each letter, otherwise it may be not be so easy.

2) Invoking the compiler: before your program can run, it must be converted into byte-code; that is the task of the compiler after it checks the syntax.

3) Executing the program: if your code compiles without errors you can then execute it.

When you choose Project/Execute, Netbeans compiles and executes your code automatically.

If you have not done so yet, adjourn to the keyboard to create and run this robot greeter program. Then, mysteriously you will have crossed the divide and can be labeled a “programmer” -- writing (or even modifying) a class is something 99% of the people on this planet have never done! If your program had/has many errors you might turn to the “What could go wrong” section in the back of the chapter for hints.

The Second Example of a Class -- A Personalized RobotGreeter

This section presents a class that is able to say “Greetings, Spike.”, or “Greetings, Buffy.”, or “Greetings, whomever.”, depending on who you ask it to greet.

Most Java applications we write will have a class with a main() method and little else. The main() method creates one or more objects and sends them messages; these objects and classes do the work of the program. This example, and the rest in this text, will have a class which only exists to interface with the outside world.
A Prototype
As you may recall, if a system is complex, build a prototype first; after that works, then (and only then!) elaborate it. Listing 2.3 is a prototype with the desired structure that always says “Greetings” (just like the last example).

Listing 2.3: Prototype Personalized RobotGreeter Class With Main Driver Program

```
1  class PersonalGreeter {
2     public void sayHi() {
3         System.out.println("Greetings!");
4     }
5  }
6
7  public class Main {
8     public static void main (String [] args) {
9         new PersonalGreeter().sayHi();
10    }
11 }
```

Notice that the `PersonalGreeter sayHi()` method does exactly what the old `main()` method did, namely displays “Greetings!” on the screen by using the idiom:

```
System.out.println("Greetings!");
```

There are two classes in Listing 2.3, they are described one by one next.

Listing 2.3 (reprise): PersonalGreeter Class Definition

```
1  class PersonalGreeter {
2     public void sayHi() {
3         System.out.println("Greetings!");
4     }
5  }
```

**Line 1:** The `PersonalGreeter` class heading. Notice that identifiers consisting of more than one word have the first letter of the second and subsequent words capitalized.

**Lines 1-5:** Next comes a block (a pair of curly brackets and what’s between them). Notice how similar it is to the `Greeter` class.

**Line 2:** Begins the `sayHi()` method. Unlike the `main()` method, here there is nothing in the ()’s.

**Line 3:** The “body”, or script, of the `sayHi()` method follows this heading. Again, like the script of every method, it is enclosed in ()’s.
Listing 2.3 (reprise, continued): Main Class Definition

```java
7     public class Main {
8         public static void main (String [] args) {
9             new PersonalGreeter().sayHi();
10        }
11    }
```

Line 9: creates a new PersonalGreeter and sends it the sayHi() message. Recall that information processing in Java is accomplished by sending messages. You can tell that sayHi() is a message by the ()s.

This prototype can only respond to one message. And its response is always to say “Greetings!”. This program is useful to convince yourself that you can write a class and make it do something (You did type in that program and test it didn’t you? If not, it’s time to go to the screen and modify your previous program, just change it so that it is like the above (but, do not change the name of the public class, or it will complain about the filename). Remember, programming is like juggling; can’t really learn by watching, can’t really learn by reading. Plus, aren’t you bored with reading these details?).


**Elaborating the Prototype Personalized Greeter**

Welcome back. Finally, let’s elaborate our prototype to make it capable of greeting various people. The question, as always, is, “How best to accomplish that?”. Usually there is more than one way to approach a program. We could add a bunch of methods like sayHiToSpike() and sayHiToBuffy(), but with the use of parameters we could add a single method that could greet whoever we wanted it to. Listing 2.4 does just that. It is an extension of the previous. The changes are in bold. The rest is unchanged.

Listing 2.4: PersonalizedGreeter Class With Main Driver Program

```java
1    class PersonalGreeter {
2        public void sayHi(String who) {
3            System.out.println("Greetings " + who + "!");
4        }
5    }
6
7    public class Main {
8       public static void main (String [] args) {
9           new PersonalGreeter().sayHi("Spike");
10           new PersonalGreeter().sayHi("Buffy");
11       }
12    }
```

Line 1: There is no public in front of class. Java only allows one public class per file.
Free advice: although the access modifiers, public, private and protected are valuable in complex programs, if you simply omit them, nothing will go wrong at this stage. Feel free to leave them out.

The two lines that are changed in the main method both send the PersonalGreeter class the SayHi message, first with the parameter "Spike", and then with the parameter "Buffy". When the parameter is "Spike" this message causes the PersonalGreeter to display “Greetings Spike!”, and when the parameter is "Buffy", it displays "Greetings Buffy!". The next section will explain Strings and parameter passing, for now it is enough to remember that things in parentheses after message names are parameters.

Quick question: How could you modify this program to greet other people?

The String Type.

You will often work with literal strings of characters, like “Java”, or “Hello world!””. There is a class in Java called String, which is designed for just that (Notice the capital S in String; you remember what that means? Right, that String is the name of a class

Details

Perhaps you are not in the mood to take on a bunch more detail, but would rather come back and grapple with it later. That would be fine. If you’d rather skip ahead to run this program first, that would be fine (it’s in the paragraph titled “Running the personalized robot greeter”, in a few pages).

Listing 2.4 (reprise): The sayHi(String) Method in the PersonalGreeter Class

```java
1    class PersonalGreeter {
2        public void sayHi(String who) {
3            System.out.println("Greetings " + who + "!");
4        }
```

Line 2: The heading for the sayHi() method. It includes the access type (public), a return type (void), the name of the method (sayHi), and a parameter (String who) enclosed in parentheses. This says that the PersonalGreeter class will respond to a sayHi() message that has a String parameter.

Line 3: The body of the sayHi method is a single message statement. First comes the object the message is sent to, System.out (which shows up the the Output window). Then comes the message, println. And finally the parameter in parentheses,("Greetings " + who + "!"). This is a single String parameter with three parts:

1. the String "Greetings "
2. the value of the parameter named who (also a String, see line 2)
3. the String "!";
Java treats the plus sign after a String as the *concatenation* operator, so it pastes those three things together to make a single String, namely, "Greetings ????!", where the ??? is the value of the String parameter was passed with the sayHi(String) message.

**Using parameters to pass information to a method.**

Every time PersonalGreeter follows this script it will display "Greetings" and "!". But, the value of the parameter, who, may be different each time the script is followed. Notice that "Greetings" and "!" are inside double quotes and so are String literals, they will be displayed literally. In contrast, who, is not in double quotes, so it will not display the letters w-h-o, but rather the current value of the parameter named who. The value of who will be whatever string was in the parentheses where the sayHi(String) message was sent to the PersonalGreeter object. There is no way to tell what that value might be without looking there.

Listing 2.4 (reprise): This sends PersonalGreeter the sayHi(String) message twice

```java
public class Main {
    public static void main (String [] args) {
        new PersonalGreeter().sayHi("Spike");
        new PersonalGreeter().sayHi("Buffy");
    }
}
```

**Line 9:** Creates a new PersonalGreeter and sends it the sayHi(String) message to with the String parameter "Spike". To repeat, this will cause the PersonalGreeter to follow (or, in technical terms, “execute”) the sayHi(String) method (script), using "Spike" as the value of the String parameter named who. Thus, inside the sayHi(String) method in PersonalGreeter,"Greetings " + who + "!", will turn into "Greetings Spike!", and be sent to System.out as the parameter with the println(String) message, and thus will end up on the screen. Many small steps. The good news is, the computer carries them out tirelessly.

**Line 10:** Exactly the same, but with the parameter value "Buffy". Line 10 was explained very slowly; ordinarily one might say instead: Line 9: Creates a PersonalGreeter and sends it the sayHi() message to with the parameter "Spike". And, to an experienced Java programmer, new PersonalGreeter().sayHi("Spike") means exactly that, and requires no explanation at all!

**A Digression**

If you have never programmed before, there are a number of new concepts in the previous example. If you did not understand it, read it again. Type it in. Get it to work. Now, read it again. Hopefully it will make more sense. If it still doesn’t, don’t panic! There are many concepts in any discipline that cannot be grasped until you have had a certain amount of experience. Let me tell you a story. When an experienced programmer is trying to learn a new computer system or language they read the entire manual or language description; it normally only takes four or five hours. Since they know little or nothing about how to use the new system or language, they can’t
understand much of what they read; thus, whenever they lose the thread, can’t understand what they are reading, they just skip to the next section. Then they go and play with the system or language for a few days or a few weeks, inventing little problems to solve. This inevitably leads to numerous unsolved mysteries and frustrations. Then, they reread the manual. This second reading is often very illuminating as now they know something of the system and have a number of questions they want to answer.

Learning to program takes time. Plus, a complete explanation of parameters is in Chapter 5. For now, patient reader, please reserve your reservations and proceed on the assumption that everything will become clearer shortly.

**Running the Personalized Robot Greeter**

Before reading on, go now to a computer. Input and run this example (or quicker, modify the Greeter class, but remember if your class is called Foo, then it must be stored in a file called Foo.java). In NetBeans you can change the name of the class by expanding first the project node (under the Projects tab), then Source Packages, then the package icon, then clicking on the name twice and, when it highlights, changing it; a panel will open, click Next, it will open another pane, click “Do Refactoring” and it will rename the file, the class, and any references to it.

After that works, insert your name and the names of several friends for the robot to greet, compile and run it again.

**Third Example: A Minimal Applet**

We’re not going to do Applets for a bit, so you can skip this; it won’t be on the quiz!

This example, the simplest possible class that extends Applet, illustrates the power of inheritance, and shows how to do graphics in the context of an Applet.

**The Power of Inheritance**

The second kind of Java program is an applet. Instead of running independently, like applications, applets run in the context of Web pages. Thus there are a number of things an applet must do to interface with the page that launches it. If you had to write the code to do those things, writing an applet would be difficult; but you don't have to. Instead, if your applet extends the built-in Java class named Applet, it can inherit all that functionality from it. So, it is very easy to write small applets by extending Applet.

Object languages, like Java, allow programmers to reuse previously written classes without having to change them, or even know what is in them, by inheritance. This is a huge improvement over languages that did not allow inheritance. Inheritance will be covered more thoroughly in Chapter 6; this chapter will simply present how to use it to write your own applets; most of the details will be postponed until you have some experience running a few programs.

**The Least You Can Do**
Before writing a robot greeter applet we will try out the simplest possible applet (you remember, first write a prototype) as shown in Listing 2.5.

Listing 2.5: A Minimal Applet

```
1    public class RobotGreeter extends java.applet.Applet {}
```

Line 1: A class declaration. It makes RobotGreeter extend java.applet.Applet; this means a RobotGreeter can do anything a java.applet.Applet can. Since there is nothing between the {}s, the RobotGreeter class does not add anything to, or change anything about the java.applet.Applet class; it is essentially another name for that class.

You might well be asking yourself if this class that appears to do nothing can run; or what it might do if it runs. Try it and see!

Executing an Applet

To run an applet (yes, run is a synonym for execute!) requires four steps: 1) create the source code, 2) compile the source code to make a .class file, 3) create HTML code to start the applet, and, 4) open the HTML code from a Java enabled browser. Fortunately Netbeans handles these details for you. See the Netbeans Appendix, “Creating the simplest applet in Netbeans”.

A Greetings Applet That Uses a Graphics Context

Now for an applet version of a robot greeter. There are a number of ways that one might make an applet that outputs "Greetings!". Listing 2.6 shows one way. This particular method uses paint(Graphics), which is the standard method for drawing in a Component, and you’ll be seeing it regularly later.

Listing 2.6: An Applet that Writes Greetings! on the Screen

```
1    public class FirstApplet extends java.applet.Applet {
2
3        /** Initialization method that will be called after the applet is
4            * loaded
5            * into the browser.
6            */
7        public void init() {
8            // TODO start asynchronous download of heavy resources
9        }
10
11       public void paint(java.awt.Graphics g) {
12          g.drawString("Greetings!", 0, 100);
13      }
```

Lines 3-5, and 7: These are comments that Netbeans wrote.
Lines 6-8: The init() method, currently with an empty method body. NetBeans writes this for you. You can add code to it as necessary (see below).

Lines 10-12: The only change to the prototype is adding the method, public void paint(Graphics); this is automatically invoked when the Applet starts.

Line 10: The method header, think of this as “abracadabra” for now.

Line 11: The only statement in paint(), draws the String "Greetings!" on the screen at location 0, 100; that’s column 0 row 100 (measured in pixels from the upper left corner). Try some other numbers.

Proceed to the screen and try this one out.

Conclusion

The first program that experienced programmers learning a new programming language write is one that simply says "Greetings", or in the older C tradition, "Hello world". This is because expert programmers have learned from painful experience that if you try to do complicated things before simple things, you can waste more hours than you might initially believe.

By writing a program that merely writes a word or two on the screen you either encounter all the difficulties of a new programming environment in the simplest possible context, where you will know to focus your attention on the environment instead of the program; or, you will discover that the programming environment is easy to use and go on to more difficult tasks immediately.

Either way, this is an example of an important problem solving (or in this case problem reducing) technique -- build a prototype first!

This chapter presented the two types of Java greetings programs and explained roughly how they were constructed. Although numerous details were presented, many others were glossed over, or explicitly ignored. The next two chapters will address some of those skipped details as well as introduce more involved examples.

What Could Go Wrong?

Any number of things! When you first program, the details are legion and any one can trip you if you forget it. The good news is that ignorance can be cured.

Compiler Error Messages.

There are many ways to generate syntax errors. Missing a single keystroke will generate an error and can generate multiple errors. Capitalization errors can be very difficult to find. Here are a few examples:

```
examples/Greeter.java [21:1] not a statement
    System.out.println("Greetings!");
```
2 errors
Errors compiling Greeter.
The errors here come from omitting the ( after println.

2 errors
Errors compiling Greeter.
The error here comes from omitting the i in public.

The error here comes from typing an s instead of an S in String.

**Review Questions**

2.1 What are the two types of Java programs. What’s the difference between them?

2.2 How (in the context of a Java program) do you print a message on the screen?

2.3 How is information processing accomplished in Java?

2.4 What are {}s used for?

2.5 When do experienced programmers write the greetings program?

2.6 What problem solving technique are they using when they do?

2.7 What are parameters used for?

2.8 Why is the S in String a capital letter?

2.9 What is public static void main(String [] args)?

**Programming Exercises**
2.10 Add these lines to your program and see what they print.

```java
System.out.println("2+2");
System.out.println(2+2);
System.out.println(2+2 == 4);
System.out.println(2 > 3);
System.out.println(2 < 3);
System.out.println("backslash-t, i.e. \t is a tab\ttab\ttab");
System.out.println("backslash-n, i.e. \n is a newline\nnewline\n");
```

2.11 Now try these

```java
int x=17;
System.out.println(x);
System.out.println("x=" + x);
System.out.println("Which of those outputs was easier to understand?");
System.out.println(x == 17);
x = x + 1;
System.out.println("after adding 1, x=" + x);
System.out.print("is x still 17? No. See? When" +
                 " it compares them it says ");
System.out.println(x == 17);
```

2.12 Modify the paint method in your applet by adding these lines.

```java
g.drawOval(200,200,100,100);
g.fillOval(400, 300, 75, 288);
g.setColor(java.awt.Color.RED);
g.drawRect(27, 27, 300, 10);
g.setColor(java.awt.Color.GREEN);
g.drawLine(30, 300, 400, 20);
```

It is likely you will need to enlarge the Applet window before you will see them all.

2.13 Now modify paint() to draw a simple picture, instead of random stuff like in the previous question. Draw a simple house. Then try your house. Then the front of the building your class meets in.