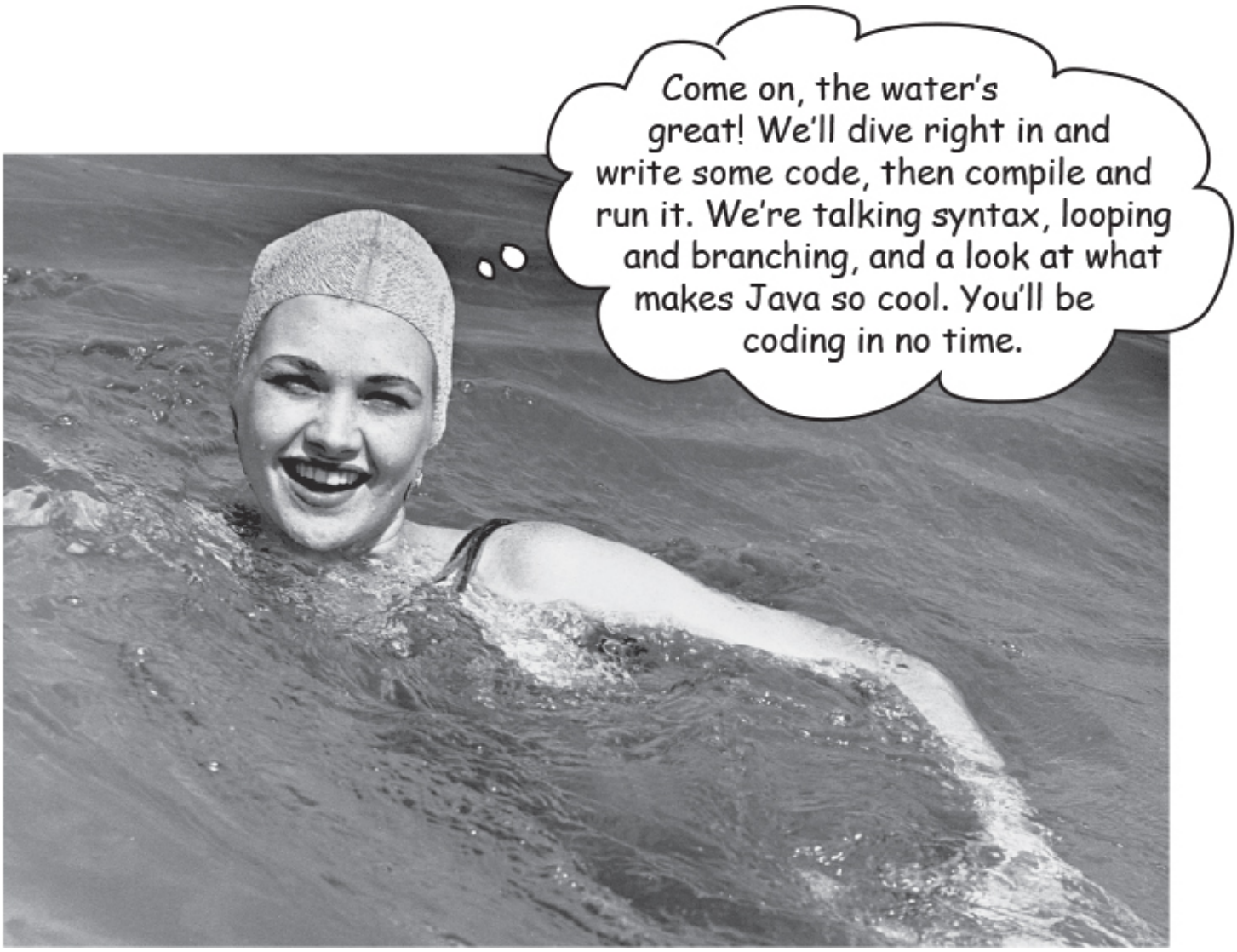
Chapter 1. Dive in A Quick Dip: Breaking the Surface



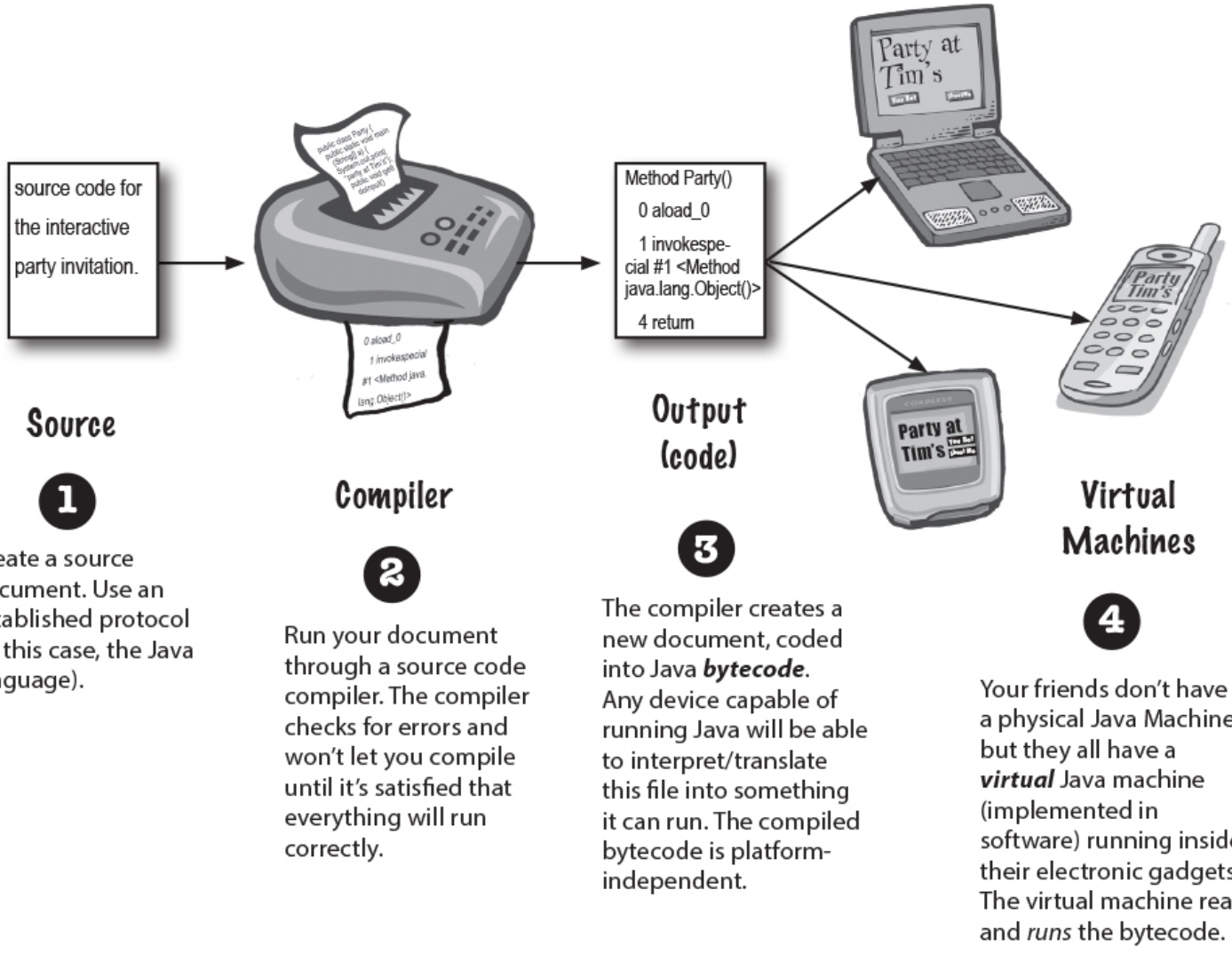
Java takes you to new places. From its humble release to the public as the (wimpy) version 1.02, Java seduced programmers with its friendly syntax, object-oriented features, memory management, and best of all —the promise of portability. The lure of **write-once/run-anywhere** is just too strong. A devoted following exploded, as programmers fought against bugs, limitations, and, oh yeah, the fact that it was dog slow. But that was ages ago. If you're just starting in Java, **you're lucky**. Some of us had to walk five miles in the snow, uphill both

ways (barefoot), to get even the most trivial applet to work. But *you*, why, *you* get to ride

the **sleeker, faster, much more powerful** Java of today.

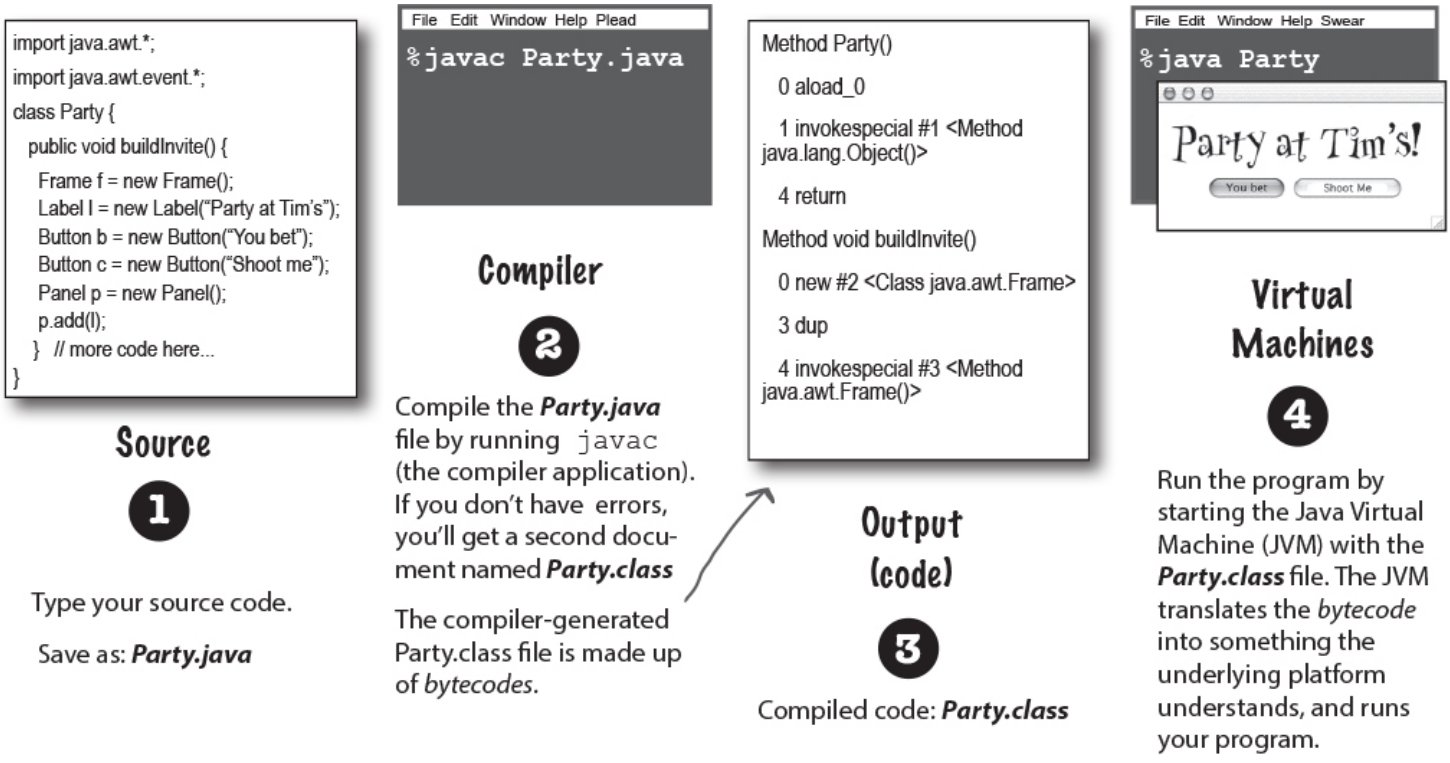
The Way Java Works

The goal is to write one application (in this example, an interactive party invitation) and have it work on whatever device your friends have.



What you'll do in Java

You'll type a source code file, compile it using the javac compiler, then run the compiled bytecode on a Java virtual machine.



NOTE

this is **NOT** meant to be a tutorial... you'll be writing real code in a moment, but for now, we just want you to get a feel for how it all fits together.

In other words, the code on this page isn't quite real, don't try to compile it..)

A Very Brief History of Java

Java was initially released (some would say "escaped"), on January 23, 1996. It's over 25 years old! In the first 25 years, Java as a language evolved, and the Java API grew enormously. The best estimate we have is that over 17 gazillion lines of Java code have been written in the last 25 years. As you spend time being a Java programmer, you will most certainly come across Java code that's quite old, and some that's much newer.

In this book we'll generally start off by using older coding styles (remember, you're likely to encounter such code in the "real world"), and then we'll introduce newer style code.

In a similar fashion, we will sometimes show you older classes in the Java API, and then show you newer alternatives.

Speed and Memory Usage

When Java was first released, it was slow. But soon after, the HotSpot VM was created, as were other performance enhancers, and while it's true that Java isn't the fastest language out there, it's considered to be a very fast language. Almost as fast as languages like C and Rust, and MUCH, MUCH faster than languages like Python or C# or even the popular new JVM language Kotlin.

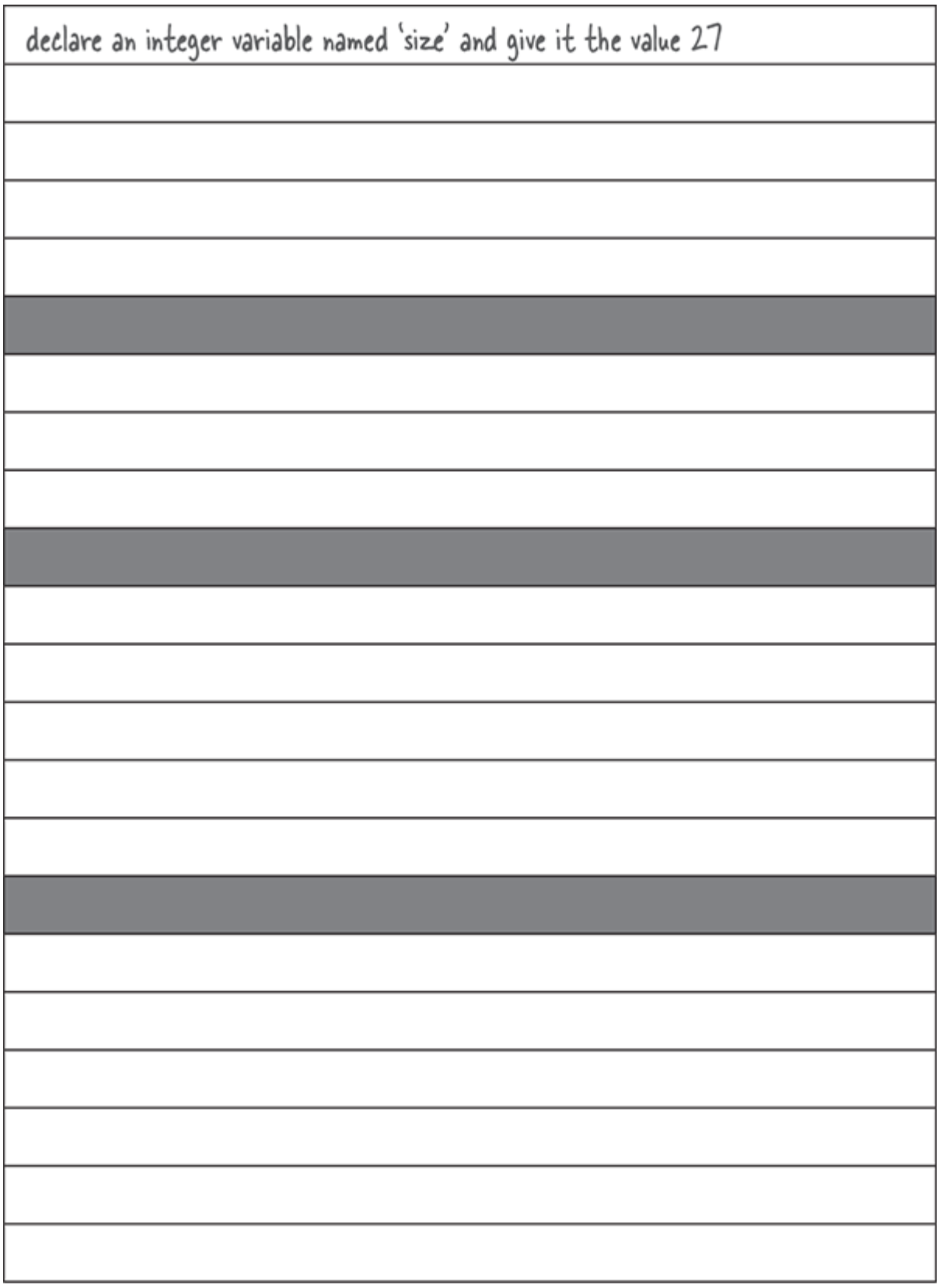
But - full disclosure - compared to C and Rust, Java uses a lot of memory.

SHARPEN YOUR PENCIL

Look how easy it is to write Java.

Try to guess what each line of code is doing... (answers are on the next page).

```
int size = 27;
String name = “Fido”;
Dog myDog = new Dog(name, size);
x = size - 5;
if (x < 15) myDog.bark(8);

while (x > 3) {
   myDog.play();
}

int[] numList = {2,4,6,8};
System.out.print(“Hello”);
System.out.print(“Dog: “ + name);
String num = “8”;
int z = Integer.parseInt(num);

try {
  readTheFile(“myFile.txt”);
}

catch(FileNotFoundException ex) {
   System.out.print(“File not found.”);
}
```



Q: The naming conventions for Java's various versions are confusing. There was JDK 1.0, and j2SE 1.2, 1.3, 1.4 then a jump to J2SE 5.0, then it changed to Java SE6, Java SE7, and last

time I checked, Java was up to Java SE 15. Can you explain what's going on?

A: It's true that for the first several years, Java's version numbers were hard to predict. But since the release of Java SE 9 in 2017, Java has been on a "rapid release" model, and it looks like from 2017 through 2020, the version numbers have become more stable and predictable.

Sharpen your pencil Answers

LOOK HOW EASY IT IS TO WRITE JAVA.

```
int size = 27;
String name = “Fido”;
Dog myDog = new Dog(name, size);
x = size - 5;
if (x < 15) myDog.bark(8);

while (x > 3){
   myDog.play();
}

int[] numList = {2,4,6,8};
System.out.print(“Hello”);
System.out.print(“Dog: “ + name);
String num = “8”;
int z = Integer.parseInt(num);

try {
  readTheFile(“myFile.txt”);
}

catch(FileNotFoundException ex) {
   System.out.print(“File not found.”);
}
```

Don’t worry about whether you understand any of this yet! Everything here is explained in great detail in the book, most within the first 40 pages). If Java resembles a language you’ve used in the past, some of this will be simple. If not, don’t worry about it. *We’ll get there...*

declare an integer variable named 'size' and give it the value 27

declare a string of characters variable named 'name' and give it the value "Fido"

declare a new Dog variable 'myDog' and make the new Dog using 'name' and 'size'

subtract 5 from 27 (value of 'size') and assign it to a variable named 'x'

if x (value of 22) is less than 15, tell the dog to bark 8 times

keep looping as long as x is greater than 3...

tell the dog to play (whatever THAT means to a dog...)

this looks like the end of the loop -- everything in { } is done in the loop

declare a list of integers variable 'numList', and put 2,4,6,8 into the list.

print out "Hello"... probably at the command-line

print out "Dog: Fido" (the value of 'name' is "Fido") at the command-line

declare a character string variable 'num' and give it the value of "8"

convert the string of characters "8" into an actual numeric value 8

try to do something...maybe the thing we're trying isn't guaranteed to work...

read a text file named "myFile.txt" (or at least TRY to read the file...)

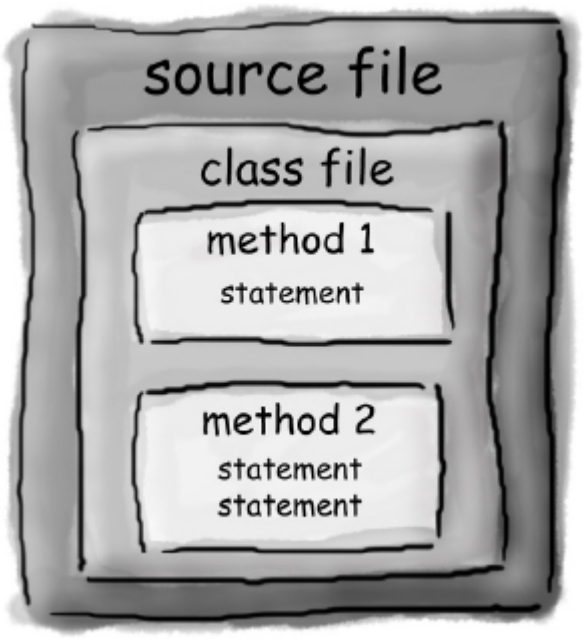
must be the end of the "things to try", so I guess you could try many things...

this must be where you find out if the thing you tried didn't work...

if the thing we tried failed, print "File not found" out at the command-line

looks like everything in the { } is what to do if the 'try' didn't work...

Code structure in Java



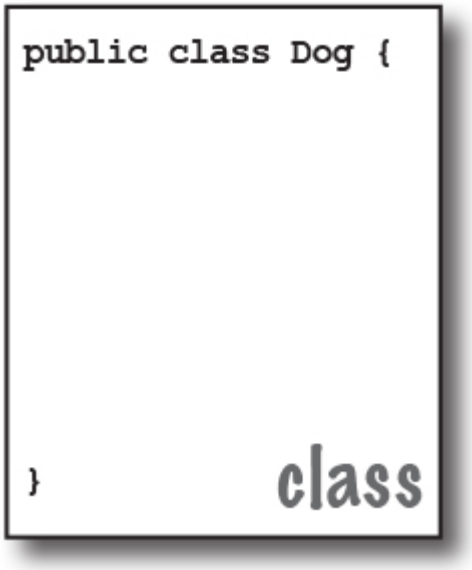
Put a class in a source file.

Put methods in a class.

Put statements in a method.

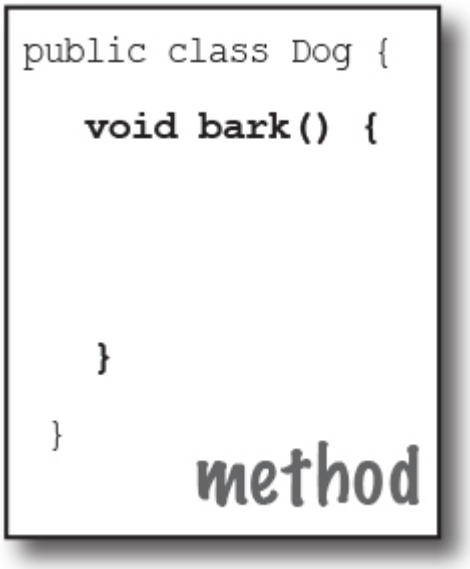
What goes in a source file?

A source code file (with the *.java* extension) holds one ***class*** definition. The class represents a *piece* of your program, although a very tiny application might need just a single class. The class must go within a pair of curly braces.



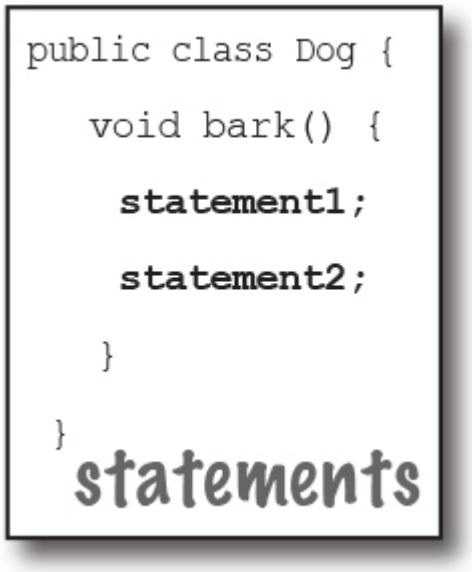
What goes in a class?

A class has one or more ***methods***. In the Dog class, the ***bark*** method will hold instructions for how the Dog should bark. Your methods must be declared *inside* a class (in other words, within the curly braces of the class).



What goes in a method?

Within the curly braces of a method, write your instructions for how that method should be performed. Method *code* is basically a set of statements, and for now you can think of a method kind of like a function or procedure.

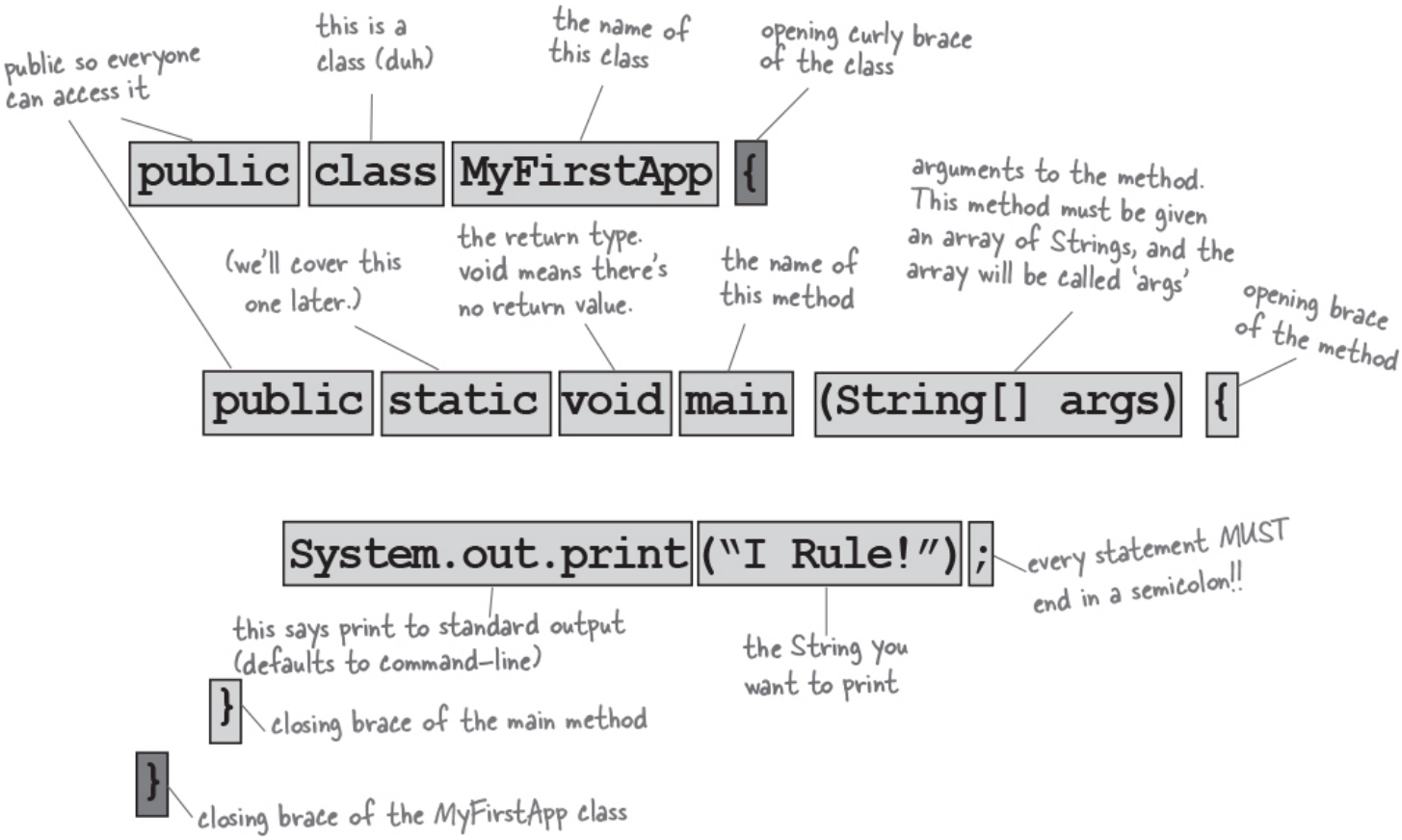


Anatomy of a class

When the JVM starts running, it looks for the class you give it at the command line. Then it starts looking for a specially-written method that looks exactly like:

```
public static void main (String[] args) {
   // your code goes here
}
```

Next, the JVM runs everything between the curly braces { } of your main method. Every Java application has to have at least one **class**, and at least one **main** method (not one main per *class*; just one main per *application*).



NOTE

Don't worry about memorizing anything right now... this chapter is just to get you started.

Writing a class with a main

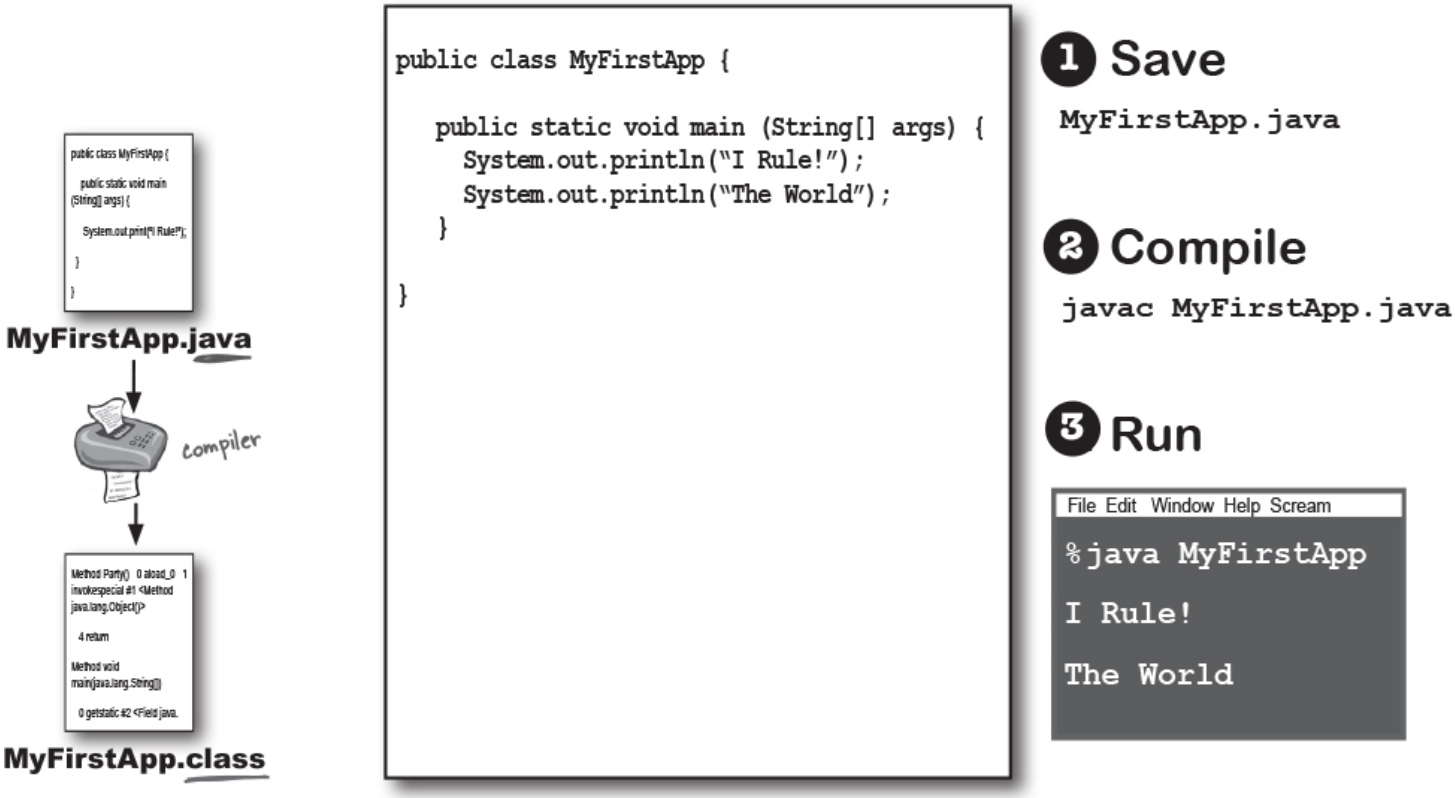
In Java, everything goes in a **class**. You'll type your source code file (with a .java extension), then compile it into a new class file (with a .class extension). When you run your program, you're really running a class.

Running a program means telling the Java Virtual Machine (JVM) to "Load the `MyFirstApp` class, then start executing its `main()` method. Keep running 'til all the code in main is finished."

In Chapter 2, we go deeper into the whole *class* thing, but for now, all you need to think is, ***how do I write Java code so that it will run?*** And it all begins with `main()`.

The `main()` method is where your program starts running.

No matter how big your program is (in other words, no matter how many *classes* your program uses), there's got to be a **main()** method to get the ball rolling.



What can you say in the main method?

Once you're inside main (or *any* method), the fun begins. You can say all the normal things that you say in most programming languages to ***make the computer do something.***

Your code can tell the JVM to:

❶ do something

Statements: declarations, assignments, method calls, etc.

```
int x = 3;
String name = "Dirk";
```

```
x = x * 17;
System.out.print(“x is ” + x);
double d = Math.random();
// this is a comment
```

❷ do something again and again

Loops: *for* and *while*

```
while (x > 12) {
   x = x - 1;
}

for (int x = 0; x < 10; x = x + 1) {
   System.out.print("x is now " + x);
}
```

❸ do something under this condition

Branching: *if/else* tests

```
if (x == 10) {
   System.out.print("x must be 10");
} else {
   System.out.print("x isn’t 10");
}
if ((x < 3) & (name.equals("Dirk"))) {
   System.out.println("Gently");
}
System.out.print("this line runs no matter what");
```


statements
loops
branching

SYNTAX FUN

- Each statement must end in a semicolon.

```
x = x + 1;
```

- A single-line comment begins with two forward slashes.

```
x = 22;
// this line disturbs me
```

- Most white space doesn't matter.

```
x = 3 ;
```

- Variables are declared with a **name** and a **type** (you'll learn about all the Java *types* in Chapter 3).

```
int weight;
//type: int, name: weight
```

- Classes and methods must be defined within a pair of curly braces.

```
public void go() {
   // amazing code here
}
```

Looping and looping and...

Java has a lot of looping constructs: *while*, *do-while*, and *for*, being the oldest. You'll get the full loop scoop later in the book, but not for awhile, so let's do *while* for now.

The syntax (not to mention logic) is so simple you're probably asleep already. As long as some condition is true, you do everything inside the loop *block*. The loop block is bounded by a pair of curly braces, so whatever you want to repeat needs to be inside that block.

The key to a loop is the *conditional test*. In Java, a conditional test is an expression that results in a *boolean* value—in other words, something that is either ***true*** or ***false***.

If you say something like, “While *iceCreamInTheTub is true*, keep scooping”, you have a clear boolean test. There either *is* ice cream in the tub or there *isn’t*. But if you were to say, “While *Bob* keep scooping”, you don’t have a real test. To make that work, you’d have to change it to something like, “While Bob is snoring...” or “While Bob is *not* wearing plaid...”

Simple boolean tests

You can do a simple boolean test by checking the value of a variable, using a comparison operator including:

< (less than)

> (greater than)

== (equality) (yes, that’s *two* equals signs)

Notice the difference between the *assignment* operator (a *single* equals sign) and the *equals* operator (*two* equals signs). Lots of programmers accidentally type = when they *want* ==. (But not you.)

```
int x = 4; // assign 4 to x
while (x > 3) {
  // loop code will run because
  // x is greater than 3
  x = x - 1; // or we’d loop forever
}
int z = 27; //
  while (z == 17) { // loop code will not run because
  // z is not equal to 17
}
```

There are no dumb Questions

Q: Why does everything have to be in a class?

A: Java is an object-oriented (OO) language. It's not like the old days when you had steam-driven compilers and wrote one monolithic source file with a pile of procedures.
In Chapter 2 you'll learn that a class is a blueprint for an object, and that nearly everything in Java is an object.

Q: Do I have to put a main in every class I write?

A: Nope. A Java program might use dozens of classes (even hundreds), but you might only have *one* with a main method — the one that starts the program running. You might write test classes, though, that have main methods for testing your *other* classes.

Q: In my other language I can do a boolean test on an integer. In Java, can I say something like:

```
int x = 1;
while (x){ }
```

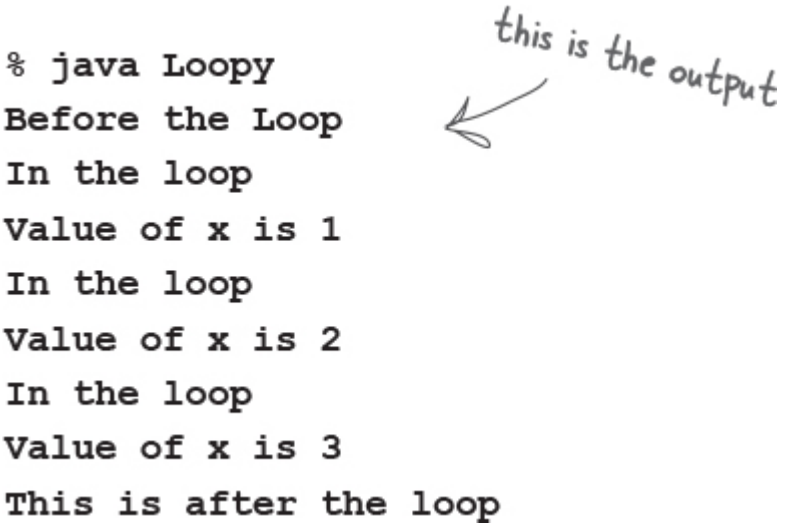
A: No. A *boolean* and an *integer* are not compatible types in Java. Since the result of a conditional test *must* be a boolean, the only variable you can directly test (without using a comparison operator) is a ***boolean.*** For example, you can say:

```
boolean isHot = true;
while(isHot) { }
```

Example of a **while** loop

```
public class Loopy {
   public static void main (String[] args) {
     int x = 1;
     System.out.println("Before the Loop");
     while (x < 4) {
       System.out.println("In the loop");
```

```
        System.out.println("Value of x is " + x);
          x = x + 1;
    }
    System.out.println("This is after the loop");
  }
}
```



BULLET POINTS

- Statements end in a semicolon ;
- Code blocks are defined by a pair of curly braces `{ }`
- Declare an *int* variable with a name and a type: **int x;**
- The **assignment** operator is *one* equals sign =
- The **equals** operator uses *two* equals signs ==
- A *while* loop runs everything within its block (defined by curly braces) as long as the *conditional test* is ***true***.
- If the conditional test is ***false***, the *while* loop code block won't run, and execution will move down to the code immediately *after* the loop block.
- Put a boolean test inside parentheses: `while` **(x == 4)** { }

Conditional branching

In Java, an *if* test is basically the same as the boolean test in a *while* loop – except instead of saying, "***while*** there's still beer...", you'll say, "***if*** there's still beer..."

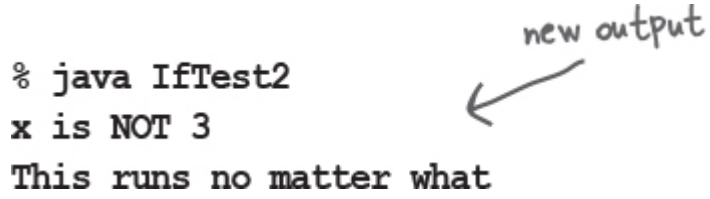
```
class IfTest {
  public static void main (String[] args) {
    int x = 3;
    if (x == 3) {
      System.out.println("x must be 3");
    }
    System.out.println("This runs no matter what");
  }
}
```

```
% java IfTest
x must be 3
This runs no matter what
```

The code above executes the line that prints "x must be 3" only if the condition (*x* is equal to 3) is true. Regardless of whether it's true, though, the line that prints, "This runs no matter what" will run. So depending on the value of *x*, either one statement or two will print out.

But we can add an *else* to the condition, so that we can say something like, "*If* there's still beer, keep coding, *else* (otherwise) get more beer, and then continue on..."

```
class IfTest2 {
  public static void main (String[] args) {
    int x = 2;
    if (x == 3) {
      System.out.println("x must be 3");
  } else {
      System.out.println("x is NOT 3");
  }
  System.out.println("This runs no matter what");
 }
}
```



SYSTEM.OUT.PRINT VS.

System.out.println

If you've been paying attention (of course you have) then you've noticed us switching between **print** and **println.**

Did you spot the difference?

System.out.***println*** inserts a newline (think of print***ln*** as **print*new*line** while System.out.***print*** keeps printing to the *same* line. If you want each thing you print out to be on its own line, use print**ln**. If you want everything to stick together on one line, use print.

SHARPEN YOUR PENCIL

Given the output:

```
% java DooBee
DooBeeDooBeeDo
```

Fill in the missing code:

```
public class DooBee {
 public static void main (String[] args) {
  int x = 1;
  while (x < _____ ) {
  System.out._________(“Doo”);
  System.out._________(“Bee”);
  x = x + 1;
  }
  if (x == ______ ) {
     System.out.print(“Do”);
  }
 }
}
```

Coding a Serious Business Application

Let’s put all your new Java skills to good use with something practical. We need a class with a *main()*, an *int* and a *String* variable, a *while* loop, and an *if* test. A little more polish, and you’ll be building that business back-end in no time. But *before* you look at the code on this page, think for a moment about how *you* would code that classic children’s favorite, “99 bottles of beer.”

```
public class BeerSong {
   public static void main (String[] args) {
     int beerNum = 99;
     String word = “bottles”;

     while (beerNum > 0) {

        if (beerNum == 1) {
          word = “bottle”; // singular, as in ONE bottle.
        }

        System.out.println(beerNum + “ ” + word + “ of beer on the wall”);
        System.out.println(beerNum + “ ” + word + “ of beer.”);
        System.out.println(“Take one down.”);
        System.out.println(“Pass it around.”);
        beerNum = beerNum - 1;

        if (beerNum > 0) {
           System.out.println(beerNum + “ ” + word + “ of beer on the wall”);
        } else {
           System.out.println(“No more bottles of beer on the wall”);
        } // end else
     } // end while loop
```

```
        } // end while loop
    } // end main method
} // end class
```

There's still one little flaw in our code. It compiles and runs, but the output isn't 100% perfect. See if you can spot the flaw, and fix it.

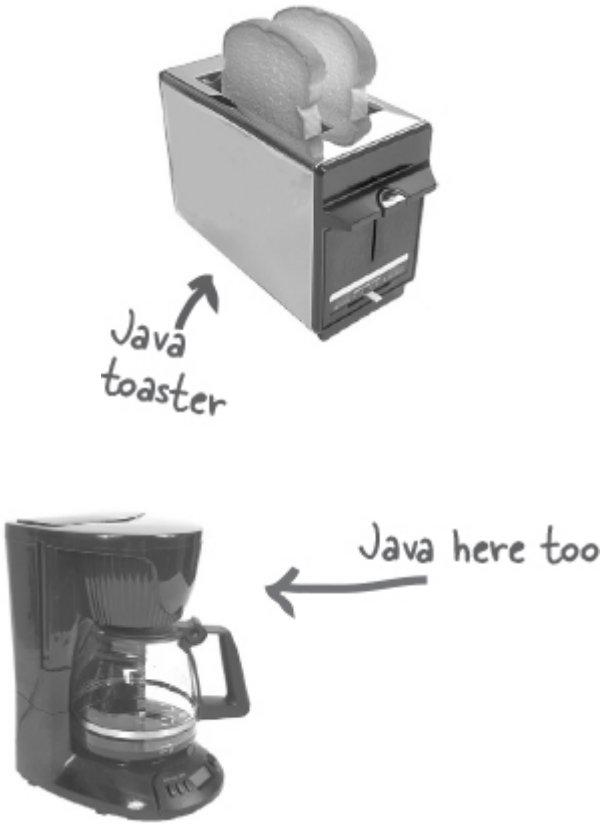
Monday Morning at Bob's Java-Enabled House

Bob's alarm clock rings at 8:30 Monday morning, just like every other weekday. But Bob had a wild weekend, and reaches for the SNOOZE button. And that's when the action starts, and the Java-enabled appliances come to life..

First, the alarm clock sends a message to the coffee maker "Hey, the geek's sleeping in again, delay the coffee 12 minutes."

The coffee maker sends a message to the Motorola™ toaster, "Hold the toast, Bob's snoozing."

The alarm clock then sends a message to Bob's Android, "Call Bob's 9 o'clock and tell him we're running a little late."



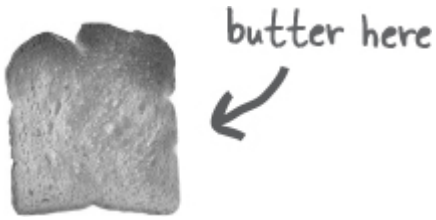
Finally, the alarm clock sends a message to Sam's (Sam is the dog) wireless collar, with the too-familiar signal that means, "Get the paper, but don't expect a walk."

A few minutes later, the alarm goes off again. And *again* Bob hits SNOOZE and the appliances start chattering. Finally, the alarm rings a third time. But just as Bob reaches for the snooze button, the clock sends the "jump and bark" signal to Sam's collar. Shocked to full consciousness, Bob rises, grateful that his Java skills and a little trip to Radio Shack™ have enhanced the daily routines of his life.

His toast is toasted.

His coffee steams.

His paper awaits.



Just another wonderful morning in ***The Java-Enabled House***.

Could this story be true? Mostly, yes!. There *are* versions of Java running in devices including cell phones (*especially* cell phones), ATMs, credit cards, home security systems, parking meters, game consoles and more –but you might not find a Java toaster or dog collar... yet.

Java Platform, Micro Edition (Java ME), allows Java applications to run on embedded and mobile devices. Java ME is very popular in the Internet of Things (IoT), world, providing security, network protocols, and all the other IoT goodies.

OK, so the beer song wasn't *really* a serious business application. Still need something practical to show the boss? Check out the Phrase-O-Matic code.

```
public class PhraseOMatic {
   public static void main (String[] args) {
```

❶ **// make three sets of words to choose from. Add your own!**

```
      String[] wordListOne = {“agnostic”, “opinionated”,
“voice activated”, “haptically driven”, “extensible”,
“reactive”, “agent based”, “functional”, “AI enabled”,
“strongly typed”};
```

```
    String[] wordListTwo = {“loosely coupled”,
“six sigma”, “asynchronous”, “event driven”, “pub
sub”, “IoT”, “cloud native”, “service oriented”,
“containerized”, “serverless”, “n-tier”, “distributed
ledger”};

    String[] wordListThree = {“framework”, “library”,
“DSL”, “REST API”, “repository”, “pipeline”, “service
mesh”, “architecture”, “perspective”, “design”,
“orientation”};
```

❷ // find out how many words are in each list

```
int oneLength = wordListOne.length;
int twoLength = wordListTwo.length;
int threeLength = wordListThree.length;
```

❸ // generate three random numbers

```
int rand1 = (int) (Math.random() * oneLength);
int rand2 = (int) (Math.random() * twoLength);
int rand3 = (int) (Math.random() * threeLength);
```

❹ // now build a phrase

```
    String phrase = wordListOne[rand1] + “ “ +
wordListTwo[rand2] + “ “ + wordListThree[rand3];
```

❺ // print out the phrase

```
    System.out.println(“What we need is a “ + phrase);
    }
}
```

NOTE

when you type this into an editor, let the code do its own word/line-wrapping! Never hit the return key when you're typing a String (a thing between "quotes") or it won't compile. So the hyphens you see on this page are real, and you can type them, but don't hit the return key until AFTER you've closed a String.

Phrase-O-Matic

How it works.

In a nutshell, the program makes three lists of words, then randomly picks one word from each of the three lists, and prints out the result. Don't worry if you don't understand *exactly* what's happening in each line. For gosh sakes, you've got the whole book ahead of you, so relax. This is just a quick look from a 30,000 foot outside-the-box targeted leveraged paradigm.

1. The first step is to create three String arrays – the containers that will hold all the words. Declaring and creating an array is easy; here's a small one:
   String[] pets = {"Fido", "Zeus", "Bin"};
   Each word is in quotes (as all good Strings must be) and separated by commas.
2. For each of the three lists (arrays), the goal is to pick a random word, so we have to know how many words are in each list. If there are 14 words in a list, then we need a random number between 0 and 13 (Java arrays are zero-based, so the first word is at position 0, the second word position 1, and the last word is position 13 in a 14-element array). Quite handily, a Java array is more than happy to tell you its length. You just have to ask. In the pets array, we'd say:
   `int x = pets.length;`
   and `x` would now hold the value 3.

NOTE

what we need here is a...

pervasive targeted process

3. We need three random numbers. Java ships out-of-the-box, off-the-shelf, shrink-wrapped, and core competent with a set of math methods (for now, think of them as functions). The `random()` method returns a random number between 0 and not-quite-1, so we have to multiply it by the number of elements (the array length) in the list we're using. We have to force the result to be an integer (no decimals allowed!) so we put in a cast (you'll get the details in Chapter 4). It's the same as if we had any floating point number that we wanted to convert to an integer:

```
int x = (int) 24.6;
```

NOTE

dynamic outside-the-box tipping-point
smart distributed core competency

---

4. Now we get to build the phrase, by picking a word from each of the three lists, and smooshing them together (also inserting spaces between words). We use the "+" operator, which *concatenates* (we prefer the more technical '*smooshes*') the String objects together. To get an element from an array, you give the array the index number (position) of the thing you want using:

```
String s = pets[0]; // s is now the String "Fido"
s = s + " " + "is a dog"; // s is now "Fido is a dog"
```

NOTE

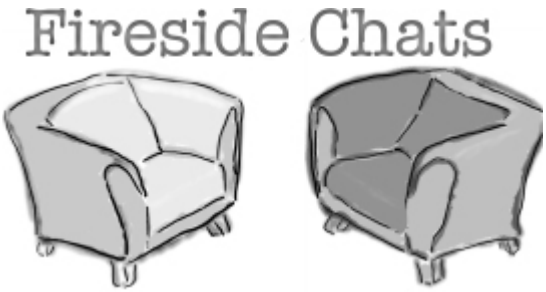
24/7 empowered mindshare
30,000 foot win-win vision

---

5. Finally, we print the phrase to the command-line and... voila! *We're in marketing*.

NOTE

six-sigma networked portal



NOTE

Tonight's Talk: **The compiler and the JVM battle over the question, "Who's more important?"**

The Java Virtual Machine

The Compiler

What, are you kidding? ***HELLO***. I am Java. I'm the guy who actually makes a program run. The compiler just gives you a file. That's it. Just a file. You can print it out and use it for wall paper, kindling, lining the bird cage whatever, but the file doesn't do anything unless I'm there to run it.

And that's another thing, the compiler has no sense of humor. Then again, if you had to spend all day checking nit-picky little syntax violations...

I don't appreciate that tone.

I'm not saying you're, like, *completely* useless. But really, what is it that you do? Seriously. I have no idea. A programmer could just write bytecode by hand, and I'd take it.

Excuse me, but without *me,* what exactly would you run? There's a *reason* Java was designed to use a bytecode compiler, for your information. If Java were a purely interpreted language, where—at runtime—the virtual machine had to translate straight-from-a-text-editor source code, a Java program would run at a ludicrously glacial pace. Java's had a challenging enough time convincing

You might be out of a job soon, buddy.

people that it's finally fast and powerful enough for most jobs.

(I rest my case on the humor thing.) But you still didn't answer my question, what *do* you actually do?

Excuse me, but that's quite an ignorant (not to mention *arrogant*) perspective. While it *is* true that — *theoretically*—you can run any properly formatted bytecode even if it didn't come out of a Java compiler, in practice that's absurd. A programmer writing bytecode by hand is like doing your word processing by writing raw postscript. And I would appreciate it if you would *not* refer to me as "buddy."

Remember that Java is a strongly-typed language, and that means I can't allow variables to hold data of the wrong type. This is a crucial safety feature, and I'm able to stop the vast majority of violations before they ever get to you. And I also—

But some still get through! I can throw ClassCast-Exceptions and sometimes I get people trying to put the wrong type of thing in an array that was declared to hold something else, and—

Excuse me, but I wasn't done. And yes, there *are* some datatype exceptions that can emerge at runtime, but some of those have to be allowed to support one of Java's other important features— dynamic binding. At runtime, a Java program can include new objects that weren't even *known* to the original programmer, so I have to allow a certain amount of flexibility. But my job is to stop anything that would never—*could* never—succeed at runtime. Usually I can tell when something won't work, for example, if a programmer accidentally tried to use a Button object as a Socket connection, I would detect that and thus protect him from causing harm at runtime.

OK. Sure. But what about *security*? Look at all the security stuff I do, and you're like, what, checking

Excuse me, but I am the first line of defense, as they say. The datatype violations I previously described could wreak havoc in a program if they were allowed to manifest. I am also the one who prevents access violations, such as code trying to invoke a private method, or change a method that

for *semicolons*? Oooohhh big security risk! Thank goodness for you!

– for security reasons – must never be changed. I stop people from touching code they're not meant to see, including code trying to access another class' critical data. It would take hours, perhaps days even, to describe the significance of my work.

Whatever. I have to do that same stuff *too*, though, just to make sure nobody snuck in after you and changed the bytecode before running it.

Of course, but as I indicated previously, if I didn't prevent what amounts to perhaps 99% of the potential problems, you would grind to a halt. And it looks like we're out of time, so we'll have to revisit this in a later chat.

Oh, you can count on it. *Buddy*.

Code Magnets

A working Java program is all scrambled up on the fridge. Can you rearrange the code snippets to make a working Java program that produces the output listed below? Some of the curly braces fell on the floor and they were too small to pick up, so feel free to add as many of those as you need!

```
if (x == 1) {
   System.out.print("d");
   x = x - 1;
}
```

```
if (x == 2) {
   System.out.print("b c");
}
```

```
class Shuffle1 {
  public static void main(String [] args) {
```

```
if (x > 2) {
   System.out.print("a");
}
```

```
int x = 3;
```

```
x = x - 1;
System.out.print("-");
```

```
while (x > 0) {
```

Output:

```
File Edit Window Help Sleep
% java Shuffle1
a-b c-d
```

BE the compiler

Each of the Java files on this page represents a complete source file. Your job is to play compiler and determine whether each of these files will compile. If they won't compile, how would you fix them?

A

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      if ( x > 3) {
        System.out.println("big x");
    }
   }
  }
}
```

B

```
public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println(“small x”);
      }
    }
}
```

C

```
class Exercise1b {
    int x = 5;
    while ( x > 1 ) {
      x = x - 1;
      if ( x < 3) {
        System.out.println(“small x”);
      }
    }
}
```

JavaCross 7.0

Let’s give your right brain something to do.

It’s your standard crossword, but almost all of the solution words are from Chapter 1. Just to keep you awake, we also threw in a few (non-Java) words from the high-tech world.

Across

4. Command-line invoker

6. Back again?

8. Can’t go both ways

9. Acronym for your laptop’s power

12. number variable type

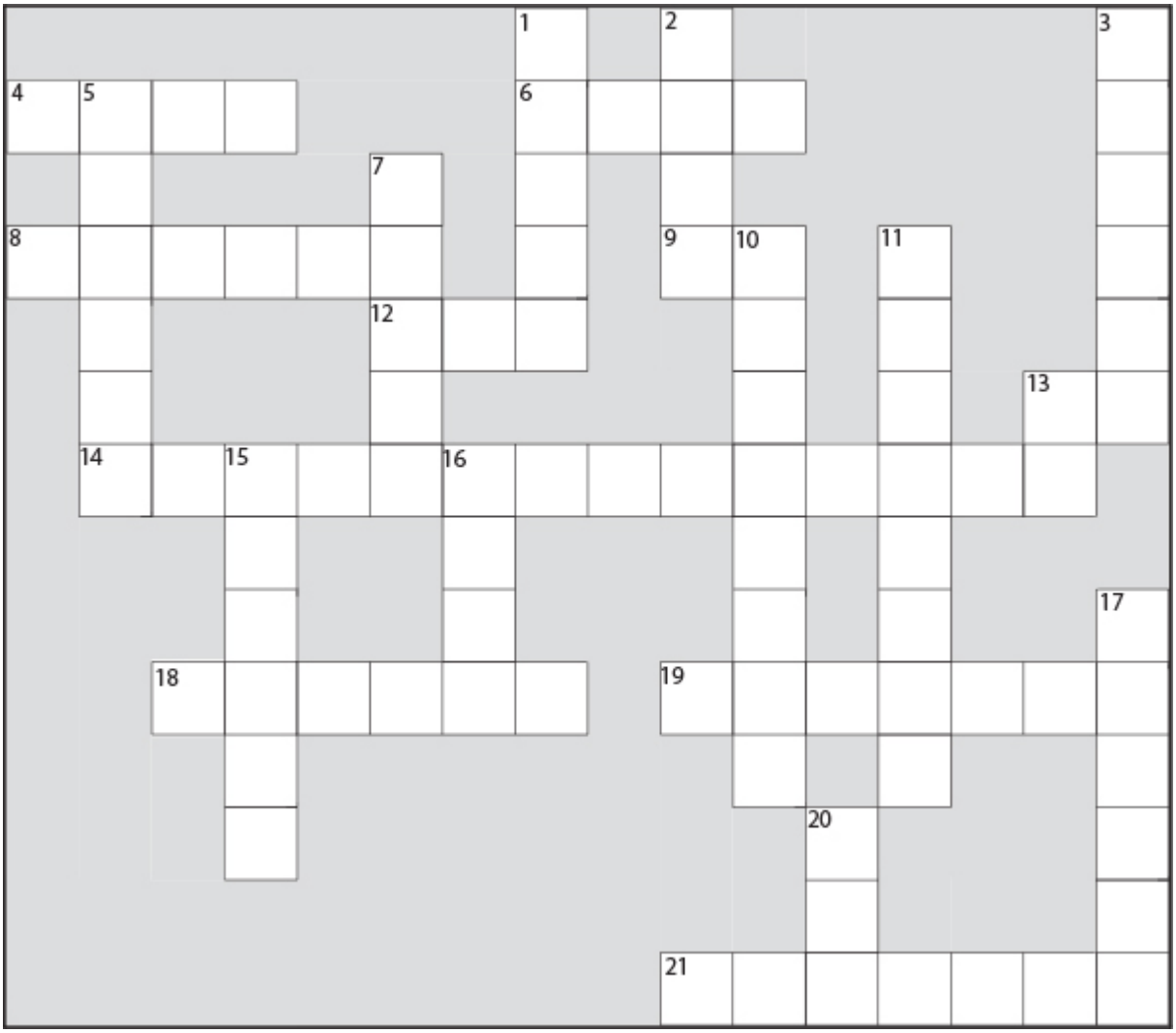
13. Acronym for a chip

14. Say something

18. Quite a crew of characters

19. Announce a new class or method

21. What’s a prompt good for?



Down

1. Not an integer (or _____ your boat)

2. Come back empty-handed

3. Open house

5. ‘Things’ holders

7. Until attitudes improve

10. Source code consumer

11. Can’t pin it down

13. Dept. of LAN jockeys

15. Shocking modifier

16. Just gotta have one

17. How to get things done

20. Bytecode consumer

A short Java program is listed below. One block of the program is missing. Your challenge is to **match the candidate block of code** (on the left), **with the output** that you'd see if the block were inserted. Not all the lines of output will be used, and some of the lines of output might be used more than once. Draw lines connecting the candidate blocks of code with their matching command-line output. (The answers are at the end of the chapter).

```
class Test {
  public static void main(String [] args) {
    int x = 0;
    int y = 0;
    while ( x < 5 ) {
      [          ]
      System.out.print(x + "" + y +" ");
      x = x + 1;
    }
  }
}
```

candidate code goes here

match each candidate with one of the possible outputs

Candidates:

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if( y > 4 ) {
  y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
  x = x + 1;
  if ( y < 3 ) {
    x = x - 1;
  }
}
y = y + 2;
```

Possible output:

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

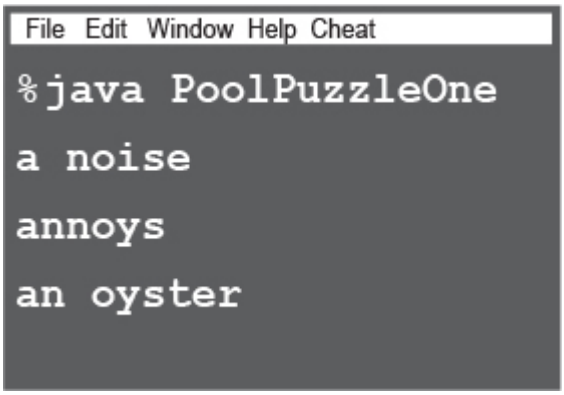
```
00 11 23 36 410
```

```
02 14 25 36 47
```

Pool Puzzle

Your ***job*** is to take code snippets from the pool and place them into the blank lines in the code. You may **not** use the same snippet more than once, and you won't need to use all the snippets. Your ***goal*** is to make a class that will compile and run and produce the output listed. Don't be fooled—this one's harder than it looks.

Output

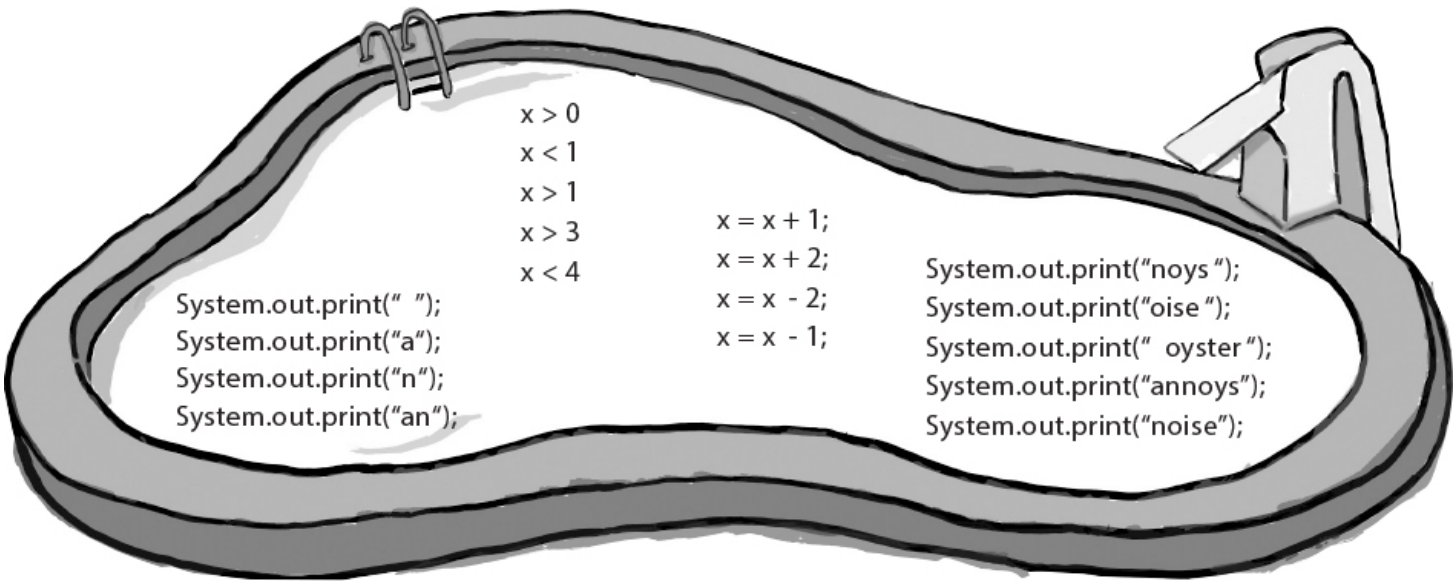



```
class PoolPuzzleOne {
  public static void main(String [] args) {
    int x = 0;

    while ( __________ ) {
      _____________________________
      if ( x < 1 ) {
       ____________________________
      }
     ____________________________
     if ( __________ ) {
```

```
        _____________________________
        ___________
      }
      if ( x == 1 ) {
        _____________________________
      }
      if ( ____________ ) {
        _____________________________
      }
      System.out.println(“”“);
      ____________
    }
  }
}
```

NOTE

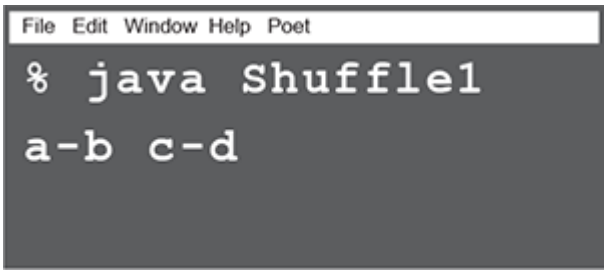
Each snippet from the pool can be used only once!



Exercise Solutions

Code Magnets:

```
class Shuffle1 {
  public static void main(String [] args) {

    int x = 3;
    while (x > 0) {

      if (x > 2) {
        System.out.print("a");
      }

      x = x - 1;
      System.out.print("-");

      if (x == 2) {
        System.out.print("b c");
      }

      if (x == 1) {
        System.out.print("d");
        x = x - 1;
      }
    }
  }
}
```




```
class Exercise1b {
  public static void main(String [] args) {
    int x = 1;
    while ( x < 10 ) {
      x = x + 1;
A     if ( x > 3) {
```

```
        System.out.println("big x");
       }
    }
  }       This will compile and run (no output), but
}         without a line added to the program, it
          would run forever in an infinite 'while' loop!
```

```
class Foo {

public static void main(String [] args) {
  int x = 5;
  while ( x > 1 ) {
    x = x - 1;
B   if ( x < 3) {
      System.out.println("small x");
    }
  }   This file won't compile without a
 }    class declaration, and don't forget
}     the matching curly brace !
```

```
class Exercise1b {
  public static void main(String [] args) {
    int x = 5;
    while ( x > 1 ) {
C     x = x - 1;
      if ( x < 3) {
        System.out.println("small x");
    }
  }     The 'while' loop code must be inside
 }      a method. It can't just be
}       hanging out inside the class.
```

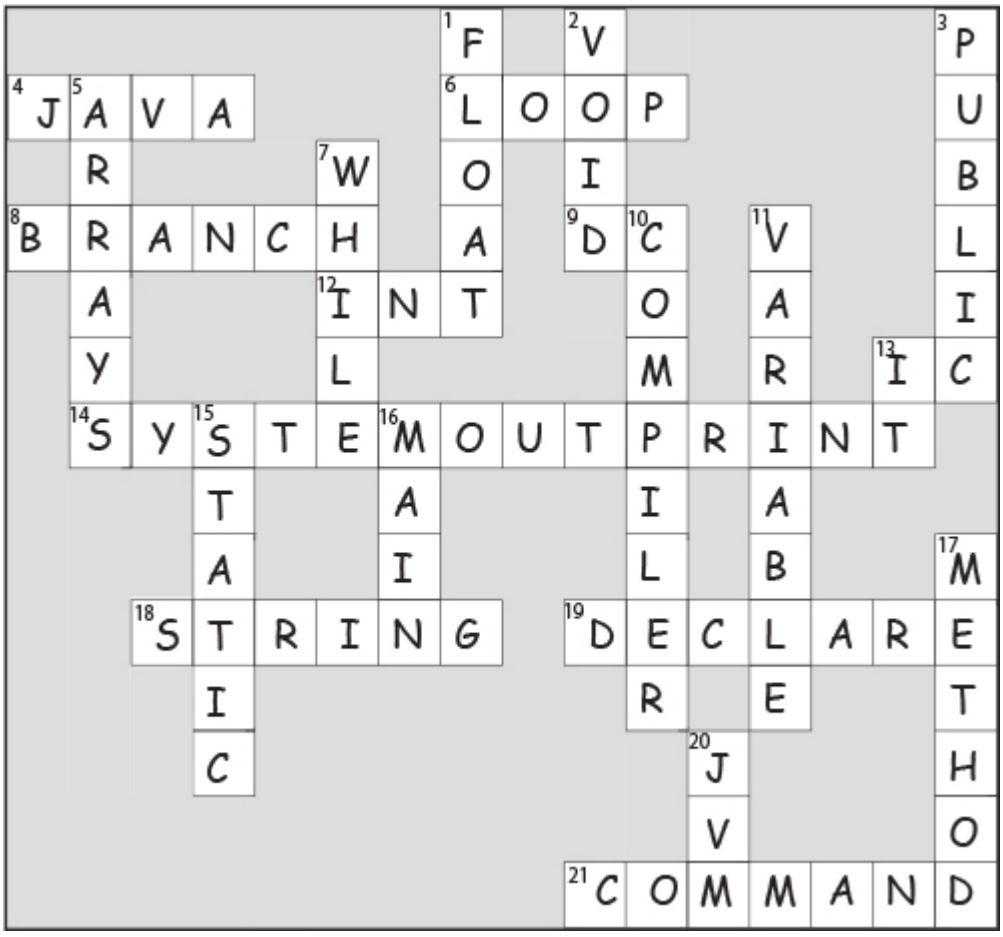
puzzle answers

```
class PoolPuzzleOne {
  public static void main(String [] args) {
    int x = 0;

    while ( X < 4 ) {

      System.out.print("a");
      if ( x < 1 ) {
         System.out.print(" ");
      }
      System.out.print("n");

      if ( X > 1 ) {

         System.out.print(" oyster");
         x = x + 2;
      }
      if ( x == 1 ) {

         System.out.print("noys");
      }
      if ( X < 1 ) {

         System.out.print("oise");
      }
      System.out.println("");

      X = X + 1;
      }
   }
}
```

```
File Edit Window Help Cheat
%java PoolPuzzleOne
a noise
annoys
an oyster
```

Free! Bonus Puzzle!

There is another way to solve the pool puzzle, that might be easier to read, can you find it?



```
class Test {
  public static void main(String [] args) {
    int x = 0;
    int y = 0;
    while ( x < 5 ) {
      [          ]

      System.out.print(x + "" + y +" ");
      x = x + 1;
    }
  }
}
```

Candidates:

```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if( y > 4 ) {
  y = y - 1;
}
```

```
x = x + 1;
y = y + x;
```

```
if ( y < 5 ) {
  x = x + 1;
  if ( y < 3 ) {
    x = x - 1;
  }
}
y = y + 2;
```

Possible output:

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```