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.

1. Create a source document. Use an established protocol (in this case, the Java language).

2. Run your document through a source code compiler. The compiler checks for errors and won’t let you compile until it’s satisfied that everything will run correctly.

3. The compiler creates a new document, coded into Java bytecode. Any device capable of running Java will be able to interpret/translate this file into something it can run. The compiled bytecode is platform-independent.

4. Your friends don’t have a physical Java Machine, but they all have a virtual Java machine (implemented in software) running inside their electronic gadgets. The virtual machine reads and runs the bytecode.
Look how easy it is to write Java.

```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.");
}
```

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

**Q:** I see Java 2 and Java 5.0, but was there a Java 3 and 4? And why is it Java 5.0 but not Java 2.0?

**A:** The joys of marketing... when the version of Java shifted from 1.1 to 1.2, the changes to Java were so dramatic that the marketers decided we needed a whole new "name," so they started calling it Java 2, even though the actual version of Java was 1.2. But versions 1.3 and 1.4 were still considered Java 2. There never was a Java 3 or 4. Beginning with Java version 1.5, the marketers decided once again that the changes were so dramatic that a new name was needed (and most developers agreed), so they looked at the options. The next number in the name sequence would be "3," but calling Java 1.5 Java 3 seemed more confusing, so they decided to name it Java 5.0 to match the "5" in version "1.5.

So, the original Java was versions 1.02 (the first official release) through 1.1 were just "Java." Versions 1.2, 1.3, and 1.4 were "Java 2." And beginning with version 1.5, Java is called "Java 5.0." But you'll also see it called "Java 5" (without the ".0") and "Tiger" (its original code-name). We have no idea what will happen with the next release...
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.

public class MyFirstApp {
    public static void main(String[] args) {
        System.out.println("I Rule!");
        System.out.println("The World");
    }
}

1. Save
   MyFirstApp.java

2. Compile
   javac MyFirstApp.java

3. Run
   java MyFirstApp
   I Rule!
   The World
Example of a while loop

```java
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");
    }
}
```

% java Loopy
Before the Loop
In the loop
Value of x is 1
In the loop
Value of x is 2
In the loop
Value of x is 3
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) { }
public class PhraseOMatic {
    public static void main (String[] args) {

        // make three sets of words to choose from. Add your own!
        String[] wordListOne = {"24/7", "multi-Tier", "30,000 foot", "B-to-B", "win-win", "front-end", "web-based", "pervasive", "smart", "six-sigma", "critical-path", "dynamic");

        String[] wordListTwo = {"empowered", "sticky", "value-added", "oriented", "centric", "distributed", 
                        "clustered", "branded", "outside-the-box", "positioned", 
                        "networked", "focused", "leveraged", "aligned", 
                        "targeted", "shared", "cooperative", "accelerated");

        String[] wordListThree = {"process", "tipping-point", "solution", "architecture", "core competency", 
                        "strategy", "mindshare", "portal", "space", "vision", 
                        "paradigm", "mission");

        // find out how many words are in each list
        int oneLength = wordListOne.length;
        int twoLength = wordListTwo.length;
        int threeLength = wordListThree.length;

        // generate three random numbers
        int rand1 = (int) (Math.random() * oneLength);
        int rand2 = (int) (Math.random() * twoLength);
        int rand3 = (int) (Math.random() * threeLength);

        // now build a phrase
        String phrase = wordListOne[rand1] + " " + wordListTwo[rand2] + " " + wordListThree[rand3];

        // print out the phrase
        System.out.println("What we need is a " + phrase);
    }
}
The Java Virtual Machine

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

The Compiler

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—

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

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 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.
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).

```java
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;
            y = x - y;
            System.out.print(y + " ");
        }
    }
}
```
Larry snuck in just moments ahead of Brad.

(Hah! So much for that foamy OO nonsense). But the smirk on Larry’s face melted when the Really Annoying Project Manager said (with that tone of disappointment), “Oh, no, that’s not how the amoeba is supposed to rotate...”

Turns out, both programmers had written their rotate code like this:

1) determine the rectangle that surrounds the shape

2) calculate the center of that rectangle, and rotate the shape around that point.

But the amoeba shape was supposed to rotate around a point on one end, like a clock hand.

“I’m toast,” thought Larry, visualizing charred Wonderbread™. “Although, hmmm. I could just add another if/else to the rotate procedure, and then just hard-code the rotation point code for the amoeba. That probably won’t break anything.” But the little voice at the back of his head said, “Big Mistake. Do you honestly think the spec won’t change again?”

Back in Larry’s cube

He figured he better add rotation point arguments to the rotate procedure. A lot of code was affected.

Testing, recompiling, the whole nine yards all over again. Things that used to work, didn’t.

```c
rotate(shapeNum, xPt, yPt) {
  // if the shape is not an amoeba,
  // calculate the center point
  // based on a rectangle,
  // then rotate
  // else
  // use the xPt and yPt as
  // the rotation point offset
  // and then rotate
```

At Brad’s laptop on his lawn chair at the Telluride Bluegrass Festival

Without missing a beat, Brad modified the rotate method, but only in the Amoeba class. He never touched the tested, working, compiled code for the other parts of the program. To give the Amoeba a rotation point, he added an attribute that all Amoebas would have. He modified, tested, and delivered (wirelessly) the revised program during a single Bela Fleck set.

```c
struct Amoeba {
  int xPoint
  int yPoint

  rotate() {
    // code to rotate an amoeba
    // using amoeba’s x and y
  }

  playSound() {
    // code to play the new
    // .hif file for an amoeba
  }
}
```
The suspense is killing me.
Who got the chair?

Amy from the second floor.
(unbeknownst to all, the Project Manager had given the spec to three programmers.)

---

**What do you like about OO?**

"It helps me design in a more natural way. Things have a way of evolving."
- Joy, 27, software architect

"Not messing around with code I've already tested, just to add a new feature."
- Brad, 32, programmer

"I like that the data and the methods that operate on that data are together in one class."
- Josh, 22, beer drinker

"Reusing code in other applications. When I write a new class, I can make it flexible enough to be used in something new, later."
- Chris, 39, project manager

"I can't believe Chris just said that. He hasn't written a line of code in 5 years."
- Daryl, 44, works for Chris

"Besides the chair?"
- Amy, 34, programmer

---

**BRAIN POWER**

**Time to pump some neurons.**

You just read a story about a procedural programmer going head-to-head with an OO programmer. You got a quick overview of some key OO concepts including classes, methods, and attributes. We'll spend the rest of the chapter looking at classes and objects (we'll return to inheritance and overriding in later chapters).

Based on what you've seen so far (and what you may know from a previous OO language you've worked with), take a moment to think about these questions:

What are the fundamental things you need to think about when you design a Java class? What are the questions you need to ask yourself?

If you could design a checklist to use when you're designing a class, what would be on the checklist?

---

**metacognitive tip**

If you're stuck on an exercise, try talking about it out loud. Speaking (and hearing) activates a different part of your brain. Although it works best if you have another person to discuss it with, pets work too. That's how our dog learned polymorphism.
Making and testing Movie objects

```java
class Movie {
    String title;
    String genre;
    int rating;

    void playIt() {
        System.out.println("Playing the movie");
    }
}
}

public class MovieTestDrive {
    public static void main(String[] args) {
        Movie one = new Movie();
        one.title = "Gone with the Stock";
        one.genre = "Tragic";
        one.rating = -2;
        Movie two = new Movie();
        two.title = "Lost in Cubicle Space";
        two.genre = "Comedy";
        two.rating = 5;
        two.playIt();
        Movie three = new Movie();
        three.title = "Byte Club";
        three.genre = "Tragic but ultimately uplifting";
        three.rating = 127;
    }
}
```

 Sharpen your pencil

<table>
<thead>
<tr>
<th>MOVIE</th>
<th>title</th>
<th>genre</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>object 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>object 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>object 3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The MovieTestDrive class creates objects (instances) of the Movie class and uses the dot operator (.) to set the instance variables to a specific value. The MovieTestDrive class also invokes (calls) a method on one of the objects. Fill in the chart to the right with the values the three objects have at the end of main().
Running the Guessing Game

```java
public class Player {
    int number = 0; // where the guess goes

    public void guess() {
        number = (int) (Math.random() * 10);
        System.out.println("I'm guessing " + number);
    }
}

public class GameLauncher {
    public static void main (String[] args) {
        GuessGame game = new GuessGame();
        game.startGame();
    }
}
```

Output (it will be different each time you run it)

```
@java GameLauncher
I'm thinking of a number between 0 and 9...
Number to guess is 7
I'm guessing 1
I'm guessing 9
I'm guessing 9
Player one guessed 1
Player two guessed 9
Player three guessed 9
Players will have to try again.
Number to guess is 7
I'm guessing 3
I'm guessing 0
I'm guessing 9
Player one guessed 3
Player two guessed 0
Player three guessed 9
Players will have to try again.
Number to guess is 7
I'm guessing 7
I'm guessing 5
I'm guessing 0
Player one guessed 7
Player two guessed 5
Player three guessed 0
We have a winner!
Player one got it right? true
Player two got it right? false
Player three got it right? false
Game is over.
```
Pool Puzzle

Your job is to take code snippets from the pool and place them into the blank lines in the code. You may use the same snippet more than once, and you won’t need to use all the snippets. Your goal is to make classes that will compile and run and produce the output listed.

Output

```
$ java EchoTestDrive
hellooo...
hellooo...
hellooo...
hellooo...
10
```

**Bonus Question!**

If the last line of output was 24 instead of 10 how would you complete the puzzle?

```
public class EchoTestDrive {
    public static void main(String[] args) {
        Echo e1 = new Echo();
        
        int x = 0;
        while (________) {
            e1.hello();
            
            if (________) {
                e2.count = e2.count + 1;
            }
        }
        if (________) {
            e2.count = e2.count + e1.count;
        }
        x = x + 1;
    }
    System.out.println(e2.count);
}
```

class _________ {
    int _______ = 0;
    void _________ {
        System.out.println("hellooo... ");
    }
}

Note: Each snippet from the pool can be used more than once!
Controlling your Dog object

You know how to declare a primitive variable and assign it a value. But now what about non-primitive variables? In other words, what about objects?

- There is actually no such thing as an object variable.
- There’s only an object reference variable.
- An object reference variable holds bits that represent a way to access an object.
- It doesn’t hold the object itself, but it holds something like a pointer. Or an address. Except, in Java we don’t really know what is inside a reference variable. We do know that whatever it is, it represents one and only one object. And the JVM knows how to use the reference to get to the object.

You can’t stuff an object into a variable. We often think of it that way... we say things like, “I passed the String to the System.out.println() method.” Or, “The method returns a Dog”, or, “I put a new Foo object into the variable named myFoo.”

But that’s not what happens. There aren’t giant expandable cups that can grow to the size of any object. Objects live in one place and one place only—the garbage collectible heap! (You’ll learn more about that later in this chapter.)

Although a primitive variable is full of bits representing the actual value of the variable, an object reference variable is full of bits representing a way to get to the object.

You use the dot operator (.) on a reference variable to say, “use the thing before the dot to get me the thing after the dot.” For example:

```java
myDog.bark();
```

means, “use the object referenced by the variable myDog to invoke the bark() method.” When you use the dot operator on an object reference variable, think of it like pressing a button on the remote control for that object.
An object reference is just another variable value.

Something that goes in a cup. Only this time, the value is a remote control.

**Primitive Variable**

```java
byte x = 7;
```

The bits representing 7 go into the variable. (00000111).

**Reference Variable**

```java
Dog myDog = new Dog();
```

The bits representing a way to get to the Dog object go into the variable. *The Dog object itself does not go into the variable!*

With primitive variables, the value of the variable is... the value (5, -26, 7, ‘a’).

With reference variables, the value of the variable is... bits representing a way to get to a specific object.

You don’t know (or care) how any particular JVM implements object references. Sure, they might be a pointer to a pointer to… but even if you know, you still can’t use the bits for anything other than accessing an object.

The 3 steps of object declaration, creation and assignment

1. **Declare a reference variable**

   ```java
   Dog myDog = new Dog();
   ```

   Tells the JVM to allocate space for a reference variable, and names that variable `myDog`. The reference variable is, forever, of type `Dog`. In other words, a remote control that has buttons to control a Dog, but not a Cat or a Button or a Socket.

2. **Create an object**

   ```java
   Dog myDog = new Dog();
   ```

   Tells the JVM to allocate space for a new Dog object on the heap (we’ll learn a lot more about that process, especially in chapter 9.)

3. **Link the object and the reference**

   ```java
   Dog myDog = new Dog();
   ```

   Assigns the new Dog to the reference variable `myDog`. In other words, programs the remote control.
Control your Dog (with a reference variable)

Dog fido = new Dog();
fido.name = "Fido";

We created a Dog object and used the dot operator on the reference variable fido to access the name variable.*

We can use the fido reference to get the dog to bark() or eat() or chaseCat().

fido.bark();
fido.chaseCat();

What happens if the Dog is in a Dog array?

We know we can access the Dog’s instance variables and methods using the dot operator, but on what?

When the Dog is in an array, we don’t have an actual variable name (like fido). Instead we use array notation and push the remote control button (dot operator) on an object at a particular index (position) in the array:

Dog[] myDogs = new Dog[3];
myDogs[0] = new Dog();
myDogs[0].name = "Fido";
myDogs[0].bark();

*Yes we know we’re not demonstrating encapsulation here, but we’re trying to keep it simple. For now, we’ll do encapsulation in chapter 4.
using references

class Dog {
    String name;
    public static void main (String[] args) {
        // make a Dog object and access it
        Dog dog1 = new Dog();
        dog1.bark();
        dog1.name = "Bart";

        // now make a Dog array
        Dog[] myDogs = new Dog[3];
        // and put some dogs in it
        myDogs[0] = new Dog();
        myDogs[1] = new Dog();
        myDogs[2] = dog1;

        // now access the Dogs using the array
        // references
        myDogs[0].name = "Fred";
        myDogs[1].name = "Marge";

        // Hmmm... what is myDogs[2] name?
        System.out.print("last dog’s name is ");
        System.out.println(myDogs[2].name);

        // now loop through the array
        // and tell all dogs to bark
        int x = 0;
        while(x < myDogs.length) {
            myDogs[x].bark();
            x = x + 1;
        }
    }
}

public void bark() {
    System.out.println(name + " says Ruff!");
}
public void eat() {
}
public void chaseCat() {
}

\section{BULLET POINTS}
\begin{itemize}
    \item Variables come in two flavors: primitive and reference.
    \item Variables must always be declared with a name and a type.
    \item A primitive variable value is the bits representing the value (5, 'a', true, 3.1416, etc.).
    \item A reference variable value is the bits representing a way to get to an object on the heap.
    \item A reference variable is like a remote control. Using the dot operator (.) on a reference variable is like pressing a button on the remote control to access a method or instance variable.
    \item A reference variable has a value of null when it is not referencing any object.
    \item An array is always an object, even if the array is declared to hold primitives. There is no such thing as a primitive array, only an array that holds primitives.
\end{itemize}