1
Learning Java

So you have decided that you would like to learn Java. Delightful. Not only that, but you have a copy of this text. Even better — you are probably exactly the kind of person we were writing it for.

We had a couple of different kinds of people in mind when we wrote this book. Most of the book is fit for every sort of reader, but certain readers will want to read some parts of this book in a different way.

Everyone

First, we want to say something to everyone, which is this: to learn to write Java programs, you must type in code and get it working. We know of no other way to learn how to program. Many have tried it an easier way, but none have succeeded.

So this book is filled with code for you to type in. As you read through this book, you will see code listings with bolded sections. Like this:

Listing 1.1 An example code listing

```
public static void main(String[] args) {
    System.out.println("Hello there ladies and gentlemen!");
}
```

Code listings with bold code in them are instructions for you to follow. So when you see bold code like this, you should immediately jump over to IntelliJ and type it in. Once you have done that, you can come back to the book, where we will explain what the code you typed in does. Eventually, you will build out an entire working program.

New To Programming

You may be relatively new to programming. If you think that is you, then you should work through every exercise from the very beginning of the book. This will get you familiar with the basic kinds of programming described in those early sections.

It is important to note that this book does not cover any of the elementary ideas of programming in any detail. If you find the process of writing down instructions and running them on a computer completely alien and confusing, you may find that this book starts a little further along the road than you. Reading through a text like *Learn Python The Hard Way* will get you further down that road. Once you do that, then you can read this book.

New To Java

You might be reading this book because you are new to Java, but you have experience with other programming languages. If you can read the following code and figure out what it does, then you are almost certainly in this category.

```
public String findLongestString(String[] strings) {
    if (strings == null || strings.length == 0) {
        return null;
    }

    String longest = strings[0];

    for (String string : strings) {
        if (longest == null || string.length() > longest.length()) {
            longest = string;
        }
    }
}
```

This does not mean that you could write a short program like this, only that you can read and understand it.

If this is you, then you do not need to spend a whole lot of time in the first portion of this book, called *Basic Programming*. This section discusses how to write basic Java programs, including how to use static methods, local variables, conditional statements, loops, and arrays.

Most of these ideas will already be familiar to you. You will want to work through the entirety of the first chapter, which covers the basics on how Java programs are written. After that, we recommend that you scan through the rest of this section without working the exercises. In particular, you will want to see how Java thinks about primitive types, and arrays.

Later on in the book, you will be writing plenty of code. Feel free to refer back to this first section if you need clarification.

People Who Do Not Want To Write Java Programs

It may be that you are reading this book because you are interested in Java, but for some reason are not interested in writing any Java code. We did not have you in mind, but welcome anyway. This book should have everything in it you would need.

If you never need to write any Java programs, then do not worry about typing in any of the bold code in the program listings. This code is there to make you into a Java programmer. So you should pay a lot of attention to it if that is something you want to do. Otherwise, you can safely skip it.

This book was written specifically for people who do not have experience writing computer programs. So if that is you, welcome! We recommend you start with the first chapter and work forward through the chapters in order, taking the time to do all of the exercises outlined. This means you should install the tools, enter the code, and execute the programs as they unfold in each chapter.

Be Prepared

One more thing before you continue: be physically prepared. Programming is something your whole body has to be there for. Your whole body does not always like to sit there and read and type things in.

So make sure you treat yourself right. Make sure you have an area free of distractions, so that you can focus on the book. A kitchen table is great for this. A time when you know you can focus without interruptions is good, too.

Make sure your brain is at rest, too. It will need to get plenty of sleep each night, so that it is ready to get cracking on all this new information you will be throwing at it. Avoid using a lot of caffeine to stay up, too. That is just like kicking your brain in the behind — it makes your brain go a lot faster, but eventually you have to do it all the time to no worthwhile end.

Get Cracking

Once you have done all that, there is only one thing left to do: Turn the page to the first chapter, and start working. Good luck.

Part I

Basic Programming

2
Getting Started

The Java language comes in two parts. The first part is the old-school part: variables, loops, conditionals, and primitive data types. The second part, the object-oriented bit, is built on top of that.

In this chapter, you will start learning to use the first part by writing a simple Java program that prints text on your screen.

The Tools

Before you get going, you will need a couple of tools: a *Java Development Kit*, or JDK, and an *integrated development environment*, or IDE.

JDK

The first thing you need is a JDK. JDK stands for *Java Development Kit*. The JDK is a kit of tools that let you write and run Java programs.

The most important parts of the JDK are `javac`, the Java compiler, and `java`, the Java runtime. To run a Java program, you first compile it into `.class` files, and then run the `.class` files using the `java` runtime program. You can use these tools directly if you like, but we will be showing how IntelliJ IDEA can take care of using them for you.

Of course, you still need to install it. Search for the Java JDK, and click the link on the Oracle web site to download it. As of this writing, the right link is `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. Once you make it there, download the Java Platform, Standard Edition JDK Version 8, and install it.

Integrated Development Environment

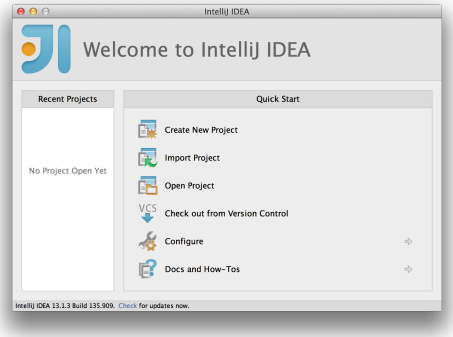
Java code, like almost all code, is written in text files. Unlike many other languages, however, it is generally agreed that Java is a language that you do not want to approach without a good IDE. The two major Java IDEs are Eclipse and IntelliJ IDEA. Eclipse was first and still beloved by many developers, but IntelliJ is newer, faster, and is full of handy ways to work with code. In this book, we will show you how to do everything with IntelliJ.

To download and install IntelliJ, go to `http://www.jetbrains.com` and follow the links to the IDEA IDE. Once you get there, download the version of the IDEA Community Edition for your operating system and install it. Community Edition is the free edition, which is enough for everything covered in this book. You should not let us stop you from paying the fine folks at IntelliJ for Ultimate if you really want to, though.

Creating A Project

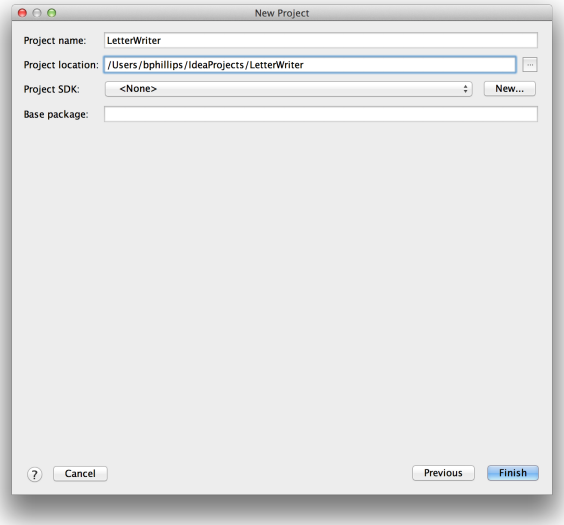
IntelliJ organizes all code into projects and modules. As a result, when you start up IntelliJ, it will assume that you want to open or create a project. So it will show a dialog that looks like this:

Figure 2.1 Opening IntelliJ



Click the Create New Project button and IntelliJ will ask what type of project you want to create:

Figure 2.2 Select Project Type



Java is highlighted, and that is what you want — a Java project. Leave Groovy unselected, and ignore the section that says Use library.

What you *will* need will be a Project SDK. To IntelliJ, an *SDK* is a set of tools that allow it to build and run a working program. In this case, it treats the JDK you downloaded earlier as a Java SDK. You will need to add an SDK to your project to get your basic program up and running.

No SDK is provided by default, and if you click the drop down box where it says <None>, you will not find any. To configure an SDK, you will need to add one by clicking on the New... button and selecting JDK from the menu that appears. IntelliJ will ask you to locate the folder your SDK lives in. On Macs, that folder will be `/Library/Java/JavaVirtualMachines/jdk1.8.0.jdk/Contents/Home`. (The `/jdk1.8.0.jdk/` part of that path may be different depending on which version of the JDK you downloaded. Use whatever you have installed.)

When you select the SDK, IntelliJ will create an SDK pointing at the JDK you navigated to and set it on your project:

Figure 2.3 SDK Now Configured

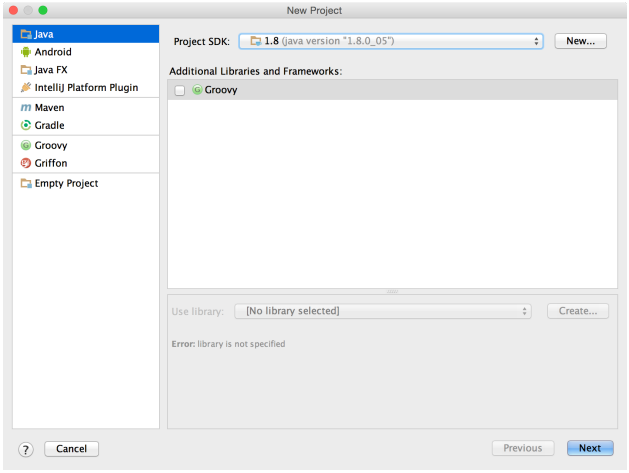
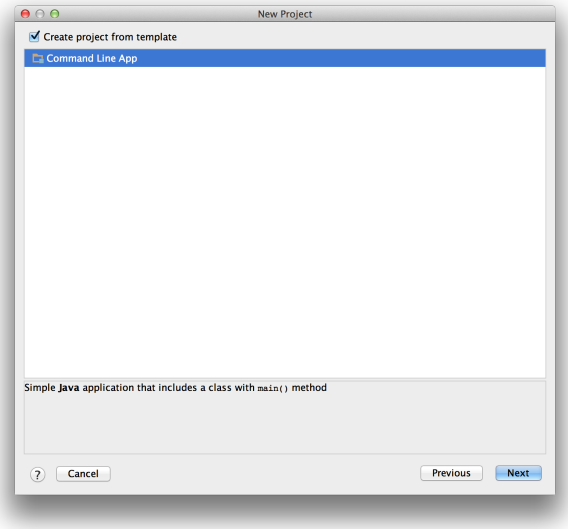


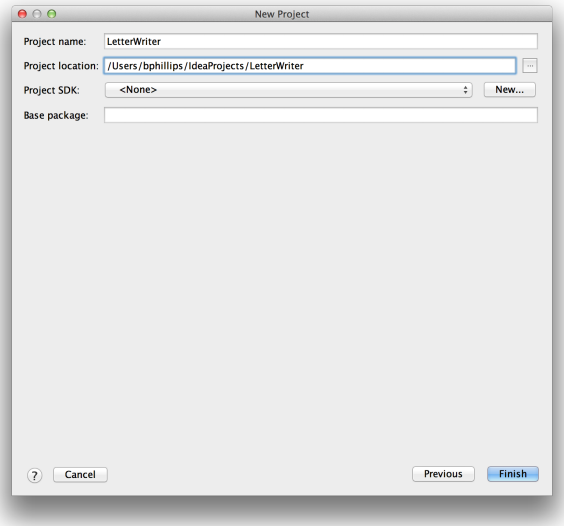
Figure 2.4 Choosing A Project Template



A template is a project with some code already written for you at the start. For this project, you will want to use a template to get going. Click the Create project from template checkbox, and then select the Command Line App template. Most of the programs you write in this book will be command line programs. The IDE will take care of running them, though, so do not worry if you are not a command line wizard.

Click Next to move on to the next section, where you will configure what your project is named and where it lives.

Figure 2.5 Configuring Your Project

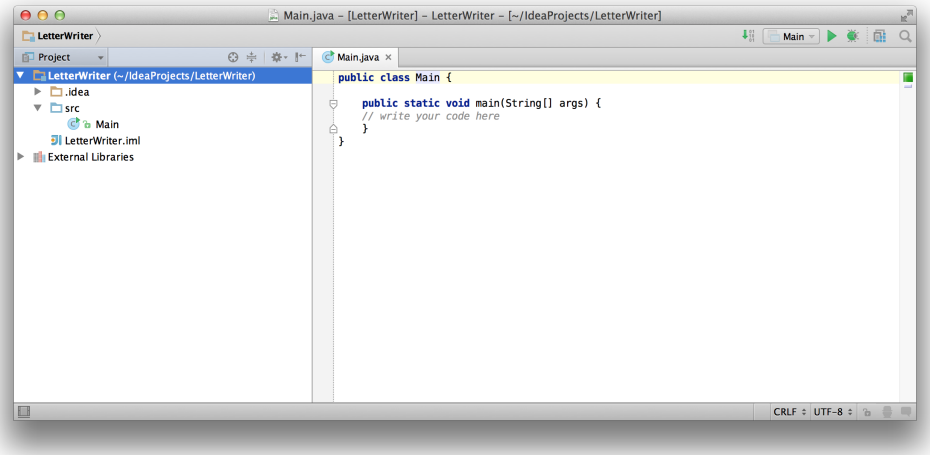


Give your project the name — "LetterWriter" — and a location. You can put your project wherever you would like it to go. This dialog also asks you to specify a Base Package. In this chapter you will focus on getting a program up and running, which does not require you to use packages. So for now, delete the default base package, leaving this field blank.

Click Finish to create your LetterWriter project. If IntelliJ asks you to create a folder, tell it to go right on ahead.

After you clear out the tool tip dialog and wait for IntelliJ to finish indexing your project, you will see your project laid out like so:

Figure 2.6 Initial Project



This is your project, with (almost) nothing in it. What you have here is your code (created for you by the template) plus some extra files that your IDE uses to keep track of everything: the `.idea` folder, `LetterWriter.iml`, and the `External Libraries` folder. You will never need to touch those extra files. Instead, focus your attention on the part of your project that will contain your program: `Main.java`.

Writing Code

In the window on the right, you will see your main source file, `Main.java`.

Figure 2.7 Template Code

```
public class Main {

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

At the top, you can see the words `public class Main`. This means that this file describes an object class called **`Main`**. Java code is required to be written inside class files, even the old-school, non-object oriented parts you will start with here. So for your purposes, this is only a place to put your **`main(String[])`**, which cannot stand on its own.

Both your class and your **`main(String[])`** are labeled `public`. This means that anyone can see and use any of this code. Until we talk about this idea in more detail later in this book, everything you write will be `public`.

Having said what you will do, your code then describes the class. Classes do a few things in Java, but for now think of them as places to put *methods*. Any methods written in between those curly braces are defined inside that class.

The next line after `public class Main` defines a method called **`main`**. A method definition has two parts: a header and a body. This first line:

```
    public static void main(String[] args)
```

...is your *header*. It is your method's visible face to the world. It says where your method can be used, what it returns, what its name is, and how you can call it. As you work through this book, more will be revealed about how to use this header. Until then, you can leave it be.

The body is then everything that goes inside the curly braces afterward. Right now, your body contains just one line:

```
    {
    // write your code here
    }
```

This is a comment in Java — anything after the `//` is ignored, so you can write whatever you want to there. For now, delete everything inside the braces and replace it with the following code:

Figure 2.8 Showing text on the screen

```
public class Main {

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

(Note the bold text — everything you are expected to type in will be bold in this book. So if you think you missed something, scan the listings for bold code.)

As you type out this code, IntelliJ will automatically show you ways that it thinks it can complete the words you are typing in. This is perfectly normal, so do not let it disturb your typing.

The line you just typed is a *statement*. Java programs are imperative and execute one line at a time, statement by statement.

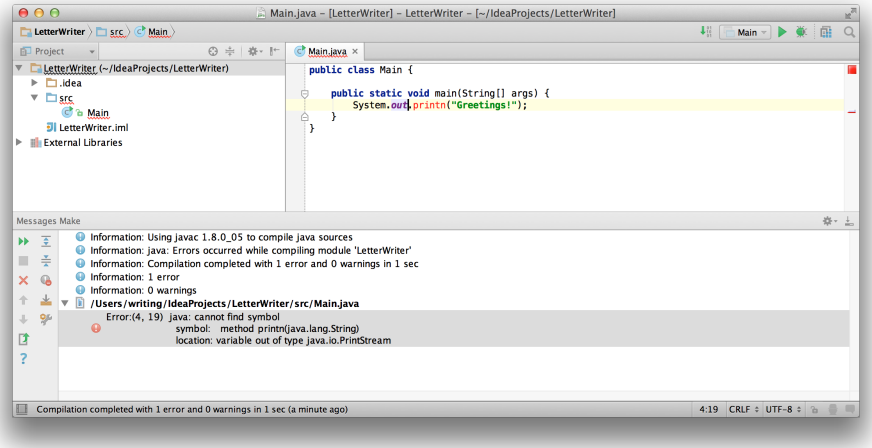
To see the effect your statement had, run your simple program. To run LetterWriter, click the green play button at the top right of IntelliJ.

When you click that button, one of two things will happen: it will work, or it will complain at you. If it works, great! If not, you have some more work to do.

Compile Errors

If IntelliJ complains, then you will see a screen that looks something like this:

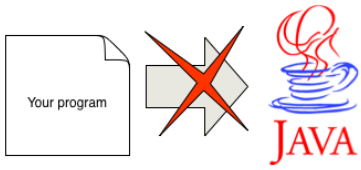
Figure 2.9 Compile Error



Eventually, you *will* see a screen that looks like this. This indicates a compile error.

Remember the two parts of your JDK: `javac`, the Java compiler, and `java`, the Java runtime. If the code you write does not make sense to `javac`, it will spit out an error and refuse to create a `.class` file out of your `Main.java`.

Figure 2.10 Failing To Compile



For the most part, these are errors that would make you go, “Woah, what?” if you were carefully proofreading the program. Calling methods, classes, or variables that do not exist; curly braces that are not closed, or where they do not make sense. In general, though, Java compile errors cannot detect errors where the program is trying to do something that will not work, like divide by zero.

Here are a few things to look out for when your program is not compiling:

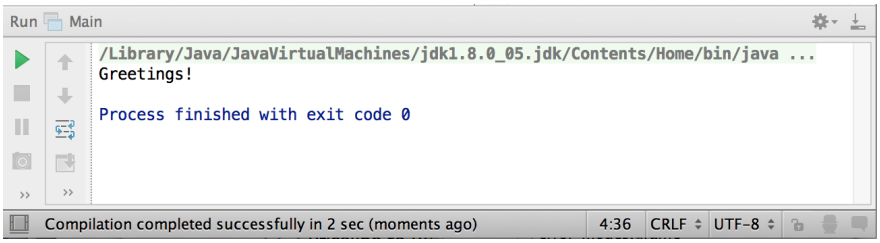
- Make sure that every open curly brace or parenthesis has a matching closed brace or paren.
- Check casing. Writing `system.out.println` will not work; `System.out.println` will.
- Check *placement*. Your method definition must be *inside* the class definition's curly braces.
- Check the kind of quotes you are using. In Java, using two single quotes is not the same as a double-quote. The example above uses double-quotes.

Running Code

Let's look more closely at what happens when you press the play button to run your code. When you hit that button, IntelliJ compiles your code, then runs it, statement by statement.

At the bottom of IntelliJ, you will see a section that looks like this:

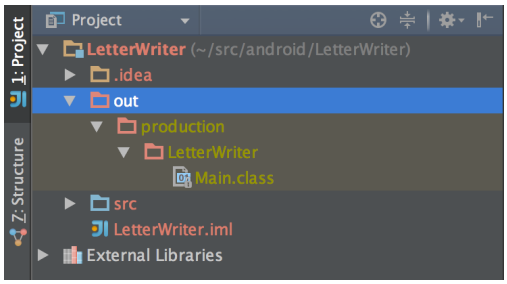
Figure 2.11 Run Tool Window



When your code runs, IntelliJ will activate and show this window, called the Run Tool Window. As you perform different tasks with IntelliJ, it will switch out the contents of this window (called the Active Tool Window) to the appropriate tool window. This tool window exists to show you the text output of your program, also called the *standard output*. There you can see that your program has shown the following text: "Greetings!"

One other thing. You read about `.class` files earlier. After running your program, you can go to the Project pane on the left and navigate down into the `out` folder to see the generated `.class` file for `Main.java`.

Figure 2.12 Seeing output class files



Method Calls

Let's look more closely at your code to see exactly how it did what it does. Your code has just one statement:

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

This statement is a *method call*. It says, "Go to the **`System`** class, find the object named `out` inside of it, and call the **`println`** method on it, passing in the string 'Greetings!' as a parameter."

Quite a mouthful, right? Let's start with the first bit:

```
System.out
```

`System` is the name of another class outside of your **`Main`** class. This class is in Java's *standard library*, the set of code that is available to use from any Java program anywhere. The **`System`** class contains methods and objects that relate to the computer your program runs on.

The next bit is not `out`, but the **`.`** in between `out` and **`System`**. This **`.`** is called the *dot operator*. The dot operator looks inside of classes and objects for other objects, methods, and (rarely) other classes. So here, the dot means, "Look for what the next word refers to inside of the **`System`** class."

So the next word, `out`, refers to something named `out` inside **`System`**. It turns out that `out` is an object that represents standard out, the text stream the Run Tool Window is displaying for you. By calling methods on this object, you can put text on that text stream, which will show them to the user.

To call a method on the `out` object, you use the dot operator one more time:

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

The last part of this statement before the semicolon, `println("Greetings!")`, is called a *method invocation*. `System.out` was merely finding the right object for the job you want to do, showing some text on the screen. The method invocation says, "Okay, standard out. Print some text for me."

The **`println`** method takes in one parameter: a *string*. Java, like most other programming languages, calls little bits of text strings. And like most other languages, Java allows you to insert a string inline inside your code by creating a double-quoted *string literal*.

The last step is to finish your statement. All statements must end with a semicolon, so that is what you do: add a semicolon at the end.

Stepping Through Code

Now that you see how one statement works, go ahead and write a few more:

Figure 2.13 More statements

```
public class Main {

    public static void main(String[] args) {
        System.out.println("Greetings!");
        System.out.println();
        System.out.println("If you like mailing checks to Nigeria,");
        System.out.println("I have an opportunity you will LOVE.");
    }
}
```

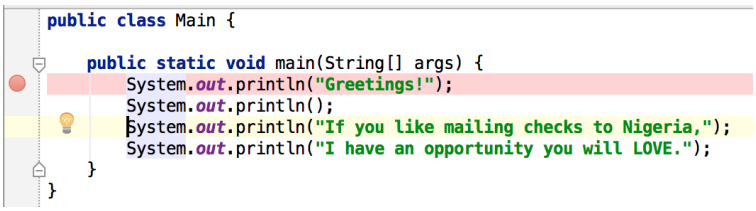
When you run this code, you will instantly see this in the active tool window:

```
Greetings!

If you like mailing checks to Nigeria,
I have an opportunity you will LOVE.
```

Your statements are being run one by one, of course. It is just happening so fast that you cannot see it happening.

To step through line by line, you can use a tool called a *debugger*. The debugger lets you inspect your code as it runs. Start by clicking to the left of the first line in your **`main`** method. Like this:

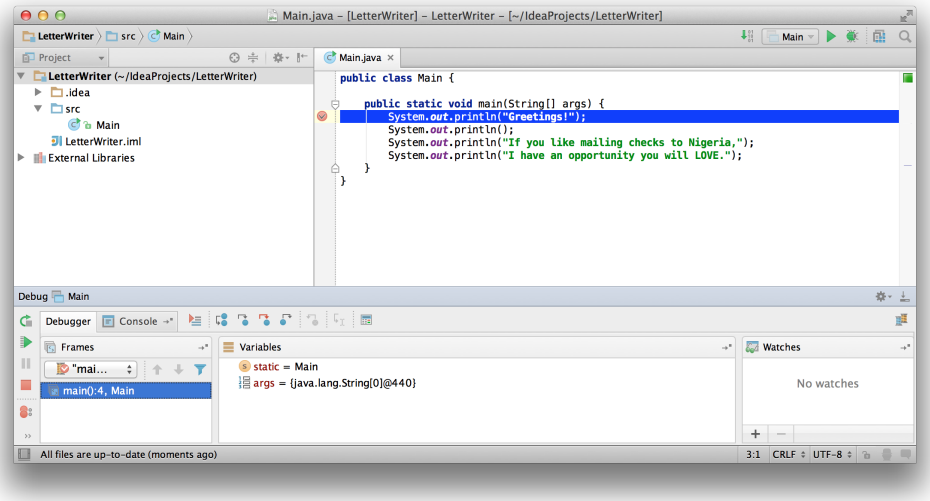
Figure 2.14 Setting a breakpoint



The red dot in the gutter on the left is a *breakpoint*. When your program is running under the debugger, anytime a breakpoint is encountered the debugger will stop everything so that you can take a look around.

To run the debugger, tap the bug icon just to the right of the green run button. (When you hover over it, it will say Debug 'Main'.) IntelliJ will run your program just like before, but this time inside the debugger. The debugger will then freeze on your breakpoint, and you will see something like this:

Figure 2.15 Stopped In The Debugger



At the top right of the screen, you can see that the red highlight on your breakpoint has turned into a bright blue. This bright blue is the current line the debugger is stopped on.

At the bottom, you can see that the active tool window has changed again. This is the Debug Tool Window, and it has a bit more going on than the Run Tool Window. If you click the Console tab, you can see the standard output of your program. The first line has not run yet, so this is currently empty except for a line showing that your debugger has successfully connected:

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_05.jdk/Contents/Home/bin/java ...
```

In the debugger, you can run just that first line of code by clicking the Step Over icon, two icons to the right of the Console tab. Stepping over a statement will run the code in that statement and move the cursor to the next line. Click the Step Over button a few more times to execute the remaining statements in your program.

When you step over the final statement, the debugger will leave your cursor on the closing brace of your method. Click on it one more time, and your program will be finished.

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_05.jdk/Contents/Home/bin/java ...
Greetings!

If you like mailing checks to Nigeria,
I have an opportunity you will LOVE.
```

For The More Curious: Whitespace

Earlier, you saw that Java can give compile time errors on bad syntax. So what about whitespace?

Like some other languages, Java does not give a hoot about whitespace. All newlines, spaces, and tab characters in between words in your program are ignored. This might lead you to believe that whitespace is not worth worrying about.

In fact, whitespace is incredibly important to the person reading your code, who you should care about far more than the compiler. The compiler ignores whitespace because people are trickier to work with than computers. If you use whitespace inconsistently, or if your program's whitespace is inconsistent with other code in the same project, you will find it impossible to go back to your program and understand what you have written.

This is because of how you read code: a lot of it is unconscious, accomplished by the many layers of your brain that work to understand what you see. When code is formatted consistently, you do not have to think about what

different parts of it mean: your unconscious mind has already figured it out. That frees up your conscious minds to do real work.

Indentation is probably the most important example of using whitespace intelligently. Your program used whitespace indentation to look like this:

```
public class Main {

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

If you indent consistently like this, you quickly gain the ability to innstantly focus on the important parts of the code, as well as spot potential issues. The following program is indented much differently, and says the same thing to Java:

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

But does it have the right number of curly braces? Not quite as easy to see, is it?

There are always two problems that a program must solve: to say something clearly to the people reading it, and to say it clearly to a computer. If you focus on using Java as your tool to solve the first problem well, you will find the second problem solves itself.

3
Using Variables And Values

In this chapter, you will write a more useful e-mail writing program. This one will write an e-mail that informs the recipient of their mortgage refinancing options. In the process, you will see how to use variables to give names to interesting values, how to use variables in expressions, and how to save and use values returned by methods you call.

Variables

Open up LetterWriter again. Start off by fixing up that initial greeting:

Figure 3.1 Make greeting nicer

```
public static void main(String[] args) {
    System.out.println("Greetings!");
    System.out.println();
    System.out.println("If you like low, low mortgage payments,");
    System.out.println("I have an opportunity you will LOVE.");
}
```

Whew! It must feel good to have the Better Business Bureau off your back.

Of course, now you need to write a better letter. To really sell this, you will want to include the new interest rate you are making available. The program should make that clear, and hopefully make it easier to find and change in the future, too.

You can solve this problem by using a *variable*. A variable is a named organizational box to put a *value* in. Values are bits of information in your code.

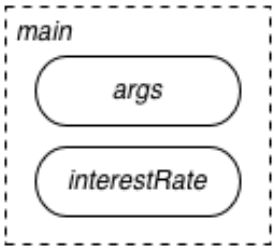
Create a variable for the interest rate by adding the following line to your program:

Figure 3.2 Add interest rate variable

```
public static void main(String[] args) {
    double interestRate;

    System.out.println("Greetings!");
    System.out.println();
    System.out.println("If you like low, low mortgage payments,");
    System.out.println("I have an opportunity you will LOVE.");
}
```

This line is a *variable declaration*. While your method is running, it has its own little dinner plate of things that it is working on. Variables that live on this dinner plate are called *local variables*.

Figure 3.3 A method's memory



Declaring a variable tells Java to make some space on your dinner plate for a bit of information, and to give it a name: `interestRate`. Variable names can be as long as you like, and may contain letters, digits, dollar signs, or underscores, but cannot start with a digit. It will live alongside the other bit of information **`main`** is holding on to, `args`. (`args` is a parameter, which is very similar to a variable. You will see more about them in Chapter 4.)

The word `double` next to the name `interestRate` is the name of the *type* of value the variable holds. Any time you define a box for your data, you must specify the type of value that box will hold.

A `double` is a double-precision floating point number value: that is, a number where you can move the position of the decimal point. Interest rates are fractional values, so a floating point number is a decent approximation for a simple program like this.

Other kinds of types for simple values (called *primitive types* in Java) are integers, which have no decimal place, booleans, which can be either true or false, and characters, which represent a single Unicode character. Here is a table of all the primitive types Java supports.

Table 3.1 Primitive Types

Types	Description
Integers: `byte`, `short`, `int`, `long`	These types are all different sizes of signed integers. A signed integer is an integer that can have a negative value. A `byte` is 8 bits (-128 to 127), a `short` is 16 bits (-32,768 to 32,767), an `int` is 32 bits (-2^{31} to $2^{31}-1$), and a `long` is 64 bits (-2^{63} to $2^{63}-1$).
Floating points: `float`, `double`	These types are both floating point values in IEEE 754 format. A `float` is a 32-bit value, and a `double` is a 64-bit value. Floating point values vary in both the size of the numbers they can represent as well as their precision. Floating point numbers are technically complex, so we cannot describe them completely here. See the end of the chapter for more on this topic.
`boolean`	May be either `true` or `false`.
`char`	A single 16-bit Unicode character. For example, `'c'` is a `char` value.

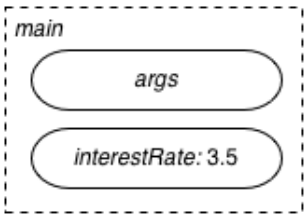
When you first create a variable, it has nothing in it. So if you try to use it, your program will fail to compile — Java does not give local variables a default value. To use your variable, you will first need to stash something in it, which you can do by *assigning* a value to it. Like so:

Figure 3.4 Assign an interest rate

```
public static void main(String[] args) {
    double interestRate;
    interestRate = 3.5;

    System.out.println("Greetings!");
    System.out.println();
    System.out.println("If you like low, low mortgage payments,");
    System.out.println("I have an opportunity you will LOVE.");
}
```

The statement `interestRate = 3.5` is an *assignment statement*. Running this statement changes the state of your dinner plate by storing a value in a variable, or another assignable location. After this statement is executed, the number `3.5` will be stored in the `interestRate` box on your dinner plate:

Figure 3.5 Saved Value



Now that you have a value stored in `interestRate`, add another line to your e-mail that uses `interestRate`'s value to print out the interest rate you are offering:

Figure 3.6 Showing your interest rate

```
    public static void main(String[] args) {
        double interestRate;
        interestRate = 3.5;

        System.out.println("Greetings!");
        System.out.println();
        System.out.println("If you like low, low mortgage payments,");
        System.out.println("I have an opportunity you will LOVE.");
        System.out.println("Interest rates have fallen to " + interestRate + "%.");
    }
```

Run this code, and you will see the interest rate you specified print right out for you.

```
Greetings!

If you like low, low mortgage payments,
I have an opportunity you will LOVE.
Interest rates have fallen to 3.5%.
```

Assignments As Statements

Assignments tell the computer to stash a value in a specific place when the assignment statement is run. So if you need a variable to have a different value later on, you can always reassign it:

Figure 3.7 Assigning twice

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

        System.out.println("If you like low, low mortgage payments,");
        System.out.println("I have an opportunity you will LOVE.");
        System.out.println("Interest rates have fallen to " + interestRate + "%.");

        interestRate = 2.5;
        System.out.println("Could they fall to " + interestRate + "?");
        System.out.println("Of course not! You should refinance immediately.");
    }
```

When you read this section of code, `interestRate` changes value between the beginning and the end of **`main(String[])`**. So depending on when you look at it, `interestRate` might mean two different things.

Assigning a new value to a variable is commonplace and useful, but often you can say the same thing by making a second variable instead. This has the advantage that your variable name only ever has one meaning.

You should do this whenever you can, because variable names are mainly there for you, not Java. Java does not really need them. It can keep track of everything on your dinner plate without any names at all, because it is a heartless machine. It usually gets rid of them and packs them down into the smallest dinner plate possible, like a plate of chicken nuggets. So if having more of them helps you express yourself more clearly, then use more of them.

Make your intentions clearer here by making a new variable called `futureInterestRate`.

Figure 3.8 Change future interest rate variable

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

    double futureInterestRate = 2.5;
    System.out.println("Could they fall to " + futureInterestRate + "?");
    System.out.println("Of course not! You should refinance immediately.");
}
```

Note that here you assigned `futureInterestRate` on the same line that you declared it. This is the most common way that variables are defined in Java, and the method you will see most frequently in this book.

You can also declare multiple variables in the same line, or declare and assign multiple variables in the same line. Do this by separating each name and assignemtn them with commas:

```
double interestRate = 3.5, futureInterestRate = 2.5, unassignedRate;
```

We will be using the declare-and-assign shorthand all the time in this book. We will avoid declaring multiple variables on the same line, though. In our practice, we find that declaring each variable on its own line is far easier to unconsciously scan.

Expressions

Now that the interest rate is written down, you need to calculate the monthly payment and assign it to another variable. To get new values besides the literal strings and doubles you have used so far, you use *expressions*.

Arithmetic Expressions

An expression is a bit of code that, after perhaps a bit of work, results in a single value. Arithmetic expressions are expressions that add, subtract, multiply, or divide values with one another to produce a new value. In short: you can write basic math using standard arithmetic notation in your code and it will work.

Let's do some math to get some intermediate values you will need.

Figure 3.9 Grab intermediate values

```
public static void main(String[] args) {
    ...
    System.out.println("If you like low, low mortgage payments,");
    System.out.println("I have an opportunity you will LOVE.");
    System.out.println("Interest rates have fallen to " + interestRate + "%.");

    int principal = 125000; // in dollars
    int numberOfYears = 30;
    int numberOfPayments = numberOfYears * 12;
    double monthlyRate = interestRate / 12 / 100; // convert from percentage

    double futureInterestRate = 2.5;
    ...
}
```

Here, you use an `int` variable named `principal` to represent dollars of principal on a loan, as well as variables named `numberOfYears` and `numberOfPayments` to keep track of intermediate values for your calculation.

All arithmetic is performed by putting together values using one of the *arithmetic operators*:

Table 3.2 Arithmetic Operators

Operator	Description
`*`	Multiply
`/`	Divide
`%`	Remainder
`+`	Add
`-`	Subtract

Order Of Evaluation

Just like in the math you learned in school, you can chain these operators together to create more complex expressions. Java knows enough about math to know its order of operations, the rules you follow when you calculate the result of a math problem.

In programming, finding the result of an expression is called *evaluating* the expression. A complex expression may be made up of several different pieces. So instead of an order of operations, there is an order of evaluation that says which pieces are evaluated first.

Look at this expression from your code. This is where you calculate the value of `monthlyRate`:

```
interestRate / 12 / 100
```

This entire bit of code is one complex, multi-part expression. To evaluate it, you evaluate its pieces from left to right. First, you evaluate each variable:

```
interestRate / 12 / 100
```

Then you evaluate the first division operator on the left:

```
3.5 / 12 / 100
```

And then the next one after that:

```
0.2917 / 100
```

To get the final value:

```
0.002917
```

Just like in arithmetic, you evaluate multiplication, division, and remainder before addition and subtraction. As you go through this book, you will learn about more kinds of expressions, and you will learn where they fit into the order of operations.

String Concatenation

In addition to adding numbers, the + operator is used to concatenate strings. Concatenating strings is putting them together — for example, if you concatenate the string “Super” with the string “man”, you get the string “Superman”. You used the + operator in Figure 3.6 to build a new string value to print to standard out:

```
System.out.println("Interest rates have fallen to " + interestRate + "%.");
```

Just like the arithmetic expression, the variables are evaluated first:

```
"Interest rates have fallen to " + interestRate + "%."
```

And then the whole expression is evaluated left to right:

`"Interest rates have fallen to " + 3.5 + "%."`

Since it is adding a string on the left with a non-string on the right, it will convert `interestRate` to a string:

`"Interest rates have fallen to " + "3.5" + "%."`

Stick them together:

`"Interest rates have fallen to 3.5" + "%."`

And then evaluate the next plus operator:

`"Interest rates have fallen to 3.5" + "%."`

To get the final value.

Multiline Comments

In this chapter and the previous chapter, you have seen and written a few single line comments:

```
int principal = 125000; // in dollars
int numberOfYears = 30;
int numberOfPayments = numberOfYears * 12;
double monthlyRate = interestRate / 12 / 100; // convert from percentage
```

You can also define multiline comments. Add a comment to explain what your **`main(String[])`** method does.

Figure 3.10 A multiline comment

```
public class Main {

    /*
    Prints out a form letter with calculated mortgage payment amounts.
     */
    public static void main(String[] args) {
        ...
    }
}
```

Anything in between `/*` and `*/` will also be ignored at runtime. Multiline comments are mainly used for documenting methods and classes — within methods, the `//` comment syntax is more common.

Method Call Expressions

Methods can return values, just like arithmetic expressions. When you call a method to get a new value, you can use that method call as a method call expression.

The final step here will be to calculate the monthly mortgage rate. The formula to do so is slightly complicated:

Figure 3.11 Mortgage payment equation

Most of that can be done using plain old multiplication and division. Java has no syntax to take the power of a number, though.

It does, however, provide a method you can use. The **`Math`** class contains a variety of useful math functions, including **`Math.pow(double,double)`**. To raise `x` to the power `y`, call `Math.pow(x, y)` as a method expression to get the result as a value.

Add one more complicated statement to get your recipient's monthly payment.

Figure 3.12 Calculate monthly payment

```
public static void main(String[] args) {
    ...
    int numberOfPayments = numberOfYears * 12;
    double monthlyRate = interestRate / 12 / 100; // convert from percentage

    double monthlyPayment = principal * (monthlyRate /
            (1 - Math.pow(1 + monthlyRate, -numberOfPayments)));

    System.out.println("You could pay as little as $" + monthlyPayment);
    System.out.println();

    double futureInterestRate = 2.5;
    System.out.println("Could they fall to " + futureInterestRate + "%?");
    System.out.println("Of course not! You should refinance immediately.");
}
```

When you run this code, you will see your program print out the following value:

```
...
You could pay as little as $561.3058597610294

Could they fall to 2.5%?
Of course not! You should refinance immediately.
```

Formatting string values nicely

While `561.3058597610294` is correct, it is not what you would want a prospective mortgage buyer to see. Your last step is to print that value rounded off to two decimal places.

The `System.out` object a method called **`format`** that solves this problem. The **`format`** method allows you to put formatting descriptions inside the string you want to print out, and then pass in the values you want to format. Use **`format`** to print out your result with the appropriate number of decimal places.

Figure 3.13 Format mortgage payment

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

    double monthlyPayment = principal * (monthlyRate /
            (1 - Math.pow(1 + monthlyRate, -numberOfPayments)));

    System.out.format("You could pay as little as $%.2f.\n\n", monthlyPayment);

    double futureInterestRate = 2.5;
    System.out.println("Could they fall to " + futureInterestRate + "%?");
    System.out.println("Of course not! You should refinance immediately.");
}
```

In the string above, the bit that says `%.2f` is called a *format specifier*. Format specifiers are only used by the **`format`** and **`printf`** methods; if you were to use the format specifier `%.2f` in a string passed in to **`println`**, it would print out `%.2f`.

Format specifiers always begin with the `%` symbol, and end with one or two letters to specify what type of value will be formatted. Here, you use the letter `f` to indicate that you are formatting a floating point value.

In between the `%` and the letters, you can also add optional flags to control how the value is printed out. Here, you use the flag `.2`, which means, "After the decimal point, print exactly two numbers."

Escape sequences

The last part of the string uses something called an *escape sequence*. Escape sequences are used to add untypeable characters to string literals.

Unlike **`println`**, **`format`** does not move on to the next line after printing what you ask it to print. The escape code "\n" will add a newline, which will make this happen.

Run your code and you will see the following letter print out:

```
Greetings!

If you like low, low mortgage payments,
I have an opportunity you will LOVE.
Interest rates have fallen to 3.5%.
You could pay as little as $561.31.

Could they fall to 2.5%?
Of course not! You should refinance immediately.
```

For The More Curious: Floating Point Decimals

The number values used in this program are informal. Nobody will be using them on tax documents or pulling money out of their bank accounts based on this math. For that reason, and because they are easier to work with for an introductory example, this code uses floating points.

It is important to understand that the floating points used here are not exact numbers, though. They are represented in base two, in binary. A decimal rendering of these numbers is often only an approximation.

To see how that works, say that you wanted to represent the fractional value `1/3`. In base ten decimals that you are so familiar with, it is impossible to represent this value. You can get close by writing `0.333333...`, but you will never get exactly `1/3`.

When converting from base ten to base two, you run into the same kind of problems. Some numbers have exact representations. For example, `0.5` can be written in base two as `0.1`. If you were to convert `0.4` into base two, though, you would get a repeating base two number: `0.011001100110...`.

This becomes a real issue when you start doing math with these values. The following line of code does not print out `0.3`:

```
System.out.println("0.1 + 0.2: " + (0.1 + 0.2));
```

It prints out this instead:

```
0.1 + 0.2: 0.30000000000000004
```

For most applications where fractional values are required, `float` and `double` are the best tool to use. Floating point math using these representations is fast and precise. It is used throughout the industry for scientific applications, graphics, audio, you name it.

If you are performing calculations that require equitable and accurate calculations in base ten, though, you need to use a different tool. In short: if you are doing math with numbers that represent dollars in someone's bank account, *do not* use `float` or `double`. Use the **`BigDecimal`** class instead. It will use a base ten decimal representation, and can round values with the correct banker's rounding method, too.

For The More Curious: Naming Conventions

There is a "hah hah only serious" joke in programming that goes like this: "There are only two hard things in computer science: cache invalidation and naming things." (And off-by-one errors.) You may never need to know what cache invalidation is, but every programmer will struggle with naming things. This is because over and over again, you will find yourself going, "Okay, I wrote this code that computes this interesting value that I want to hold on to. What should I call it?"

Answering that question is more art than science. This is a nice way of saying that people argue about it a lot, and nobody is absolutely right. There is consensus in one part of the answer, though, which is that our names for things should at least look similar to one another.

In this book, we will rarely address the topic of our naming convention. Instead, the example code you type in will exhibit our preferred naming convention. We prefer descriptive, short names. All variables and method names will be in camel case, constants will be in all caps with underscores, and classes will be in caps case. When we start discussing Android, we will shift to using some conventions popular in that community.

One last note about naming conventions: someday, you may find yourself working with other programmers or on an existing program with a lot of code. When you do, you may find that it uses a naming convention that seems bizarre, old fashioned, or even actively brain-damaged.

If you find yourself in this situation, remember: consistency is more important than perfection. Following the existing convention is almost always a better path than using your own. Have respect for those who came before you. Who knows what demons they slayed with the ugly tools they left for you?

Challenge

Gain some more practice working with values and expressions by writing code to calculate some other interesting values:

- In your mortgage calculator, show the rates for 15-year mortgages as well as 30-year mortgages.
- Next, show how much money you will pay over the lifetime of the loan.
- Solve a math problem: A cannonball is dropped from an airplane traveling horizontally at 134 meters per second, after which it accelerates downward. Each second, it accelerates by 9.8 meters per second downward. How fast is it going 12 seconds after it is dropped? (If you use the Pythagorean theorem, you can use the vertical and horizontal velocities to find the velocity of the cannonball.)

4
Static Methods

In the last chapter, you wrote a program that said, “Here is how you print out a form letter to sell a mortgage.” By programming standards, your **`main`** method was a medium-length method.

In this chapter, you will use the next organizational tool in your Java toolbelt: the static method. You will use a static method to print out more than one mortgage offer.

Repetition

The static method is a new-ish name for an old idea: giving a few lines of code a name so that they can be run repeatedly. For example, your form letter shows an old interest rate and a possible new interest rate. You could easily calculate the payment for that new interest rate, too, like this.

Figure 4.1 Parts of your program

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

    System.out.format("You could pay as little as $%.2f.\n\n", monthlyPayment);

    double futureInterestRate = 2.5;
    double futureMonthlyRate = futureInterestRate / 12 / 100;

    double futureMonthlyPayment = principal * (futureMonthlyRate /
            (1 - Math.pow(1 + futureMonthlyRate, -numberOfPayments)));

    System.out.println("Could they fall to " + futureInterestRate + "%?");
    System.out.format("Could you pay $%.2f per month instead?\n", futureMonthlyPayment);
    System.out.println("Of course not! You should refinance immediately.");
}
```

Since you are using it a second time, it makes sense to give this section of code its own name. There are a few ways to do that in Java, but the static method is the most straightforward.

Writing A Static Method

So instead of writing out another few lines of code, write out a new static method called **`getMonthlyPayment`**.

Figure 4.2 Write monthly payment method

```
public class Main {

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

    public static double getMonthlyPayment(int principal, double interestRate,
            int numberOfYears) {
        int numberOfPayments = numberOfYears * 12;
        double monthlyRate = interestRate / 12 / 100; // convert from percentage

        double monthlyPayment = principal * (monthlyRate /
                (1 - Math.pow(1 + monthlyRate, -numberOfPayments)));

        return monthlyPayment;
    }
}
```

This is your brand new method. You can call it from another section of code just like you did when you used the **`Math.pow`** method. Add some code inside **`main`** to call **`getMonthlyPayment`**.

Figure 4.3 Call your new method

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

    double futureInterestRate = 2.5;
    double futureMonthlyPayment =
        getMonthlyPayment(principal, futureInterestRate, numberOfYears);

    System.out.println("Could they fall to " + futureInterestRate + "%?");
    System.out.format("Could you pay $%.2f per month instead?\n", futureMonthlyPayment);
    System.out.println("Of course not! You should refinance immediately.");
}
```

Run your program, and you should see a new line in your output.

```
...

Could they fall to 2.5%?
Could you pay $493.90 per month instead?
Of course not! You should refinance immediately.
```

Take a closer look at your method to see how it works. Look at the first line.

```
    public static double getMonthlyPayment(int principal, double interestRate,
        int numberOfYears) {
```

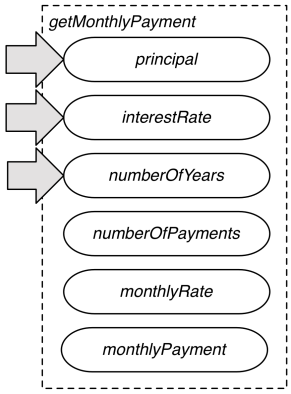
This is the method header, which we discussed earlier on in the book. The first word, the access modifier, means that your method is a *public* method. Public methods can be seen and called from any method in any class. Later on when you learn about objects, we will talk about other kinds of access modifiers and why they are useful. Until then, all of your methods will be `public`.

The second word means that your method is a *static* method. This means that your method is not related to any specific object. Again, until you start writing object classes in this book, all your methods will be static.

The third word, `double`, is the *return type* of your method. When you call a method, the value it gives back to you is called the *return value* of that method. The return type is the type of that value. So for example, the **`Math.pow`** method that you used in the previous chapter has a return type of `double`, just like **`getMonthlyPayment`**.

Finally, you have the name of the method, **`getMonthlyPayment`**, followed by a list of *parameters*. The parameters look a lot like the variables you declared in the last chapter, and for good reason. For the most part, they work the same way, too, except for one detail.

Figure 4.4 Your method's data

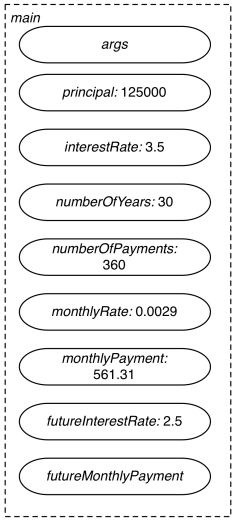


Just like a variable, a parameter is a named value where you can store things. You can assign to parameters and use them in expressions. Unlike variables, though, parameters get their initial values when you call them.

Stack Data And Control Flow

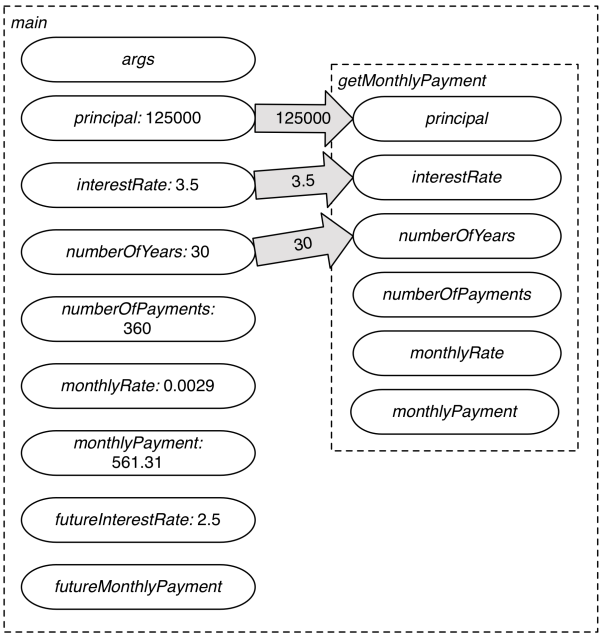
Take a look at where you call **`getMonthlyPayment`** from inside **`main`**. At that moment, **`main`**'s dinner plate looks like this:

Figure 4.5 Before Calling



When **`getMonthlyPayment`** is called, a new dinner plate needs to be created for it to live in. The new plate goes on top of **`main`**, where it is populated with values copied from **`main`**'s dinner plate.

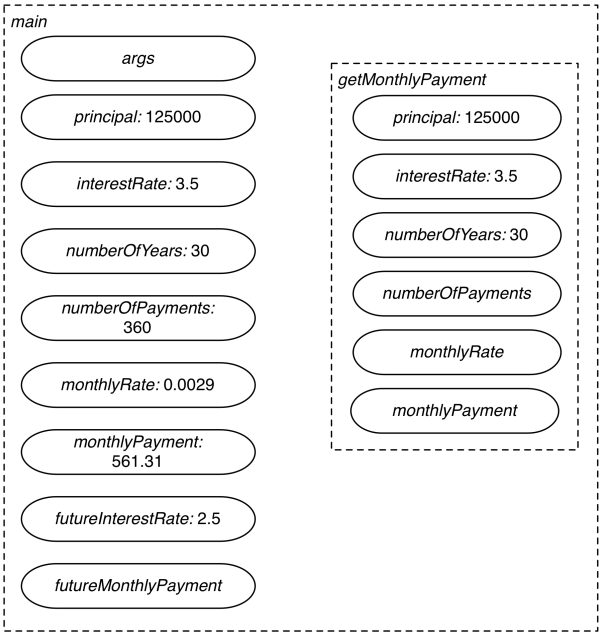
Figure 4.6 Passing Values



Notice that your old **`main`** dinner plate does not get discarded when you call the new method. Instead, the new plate goes right on top of it. This is called a *stack*. The process of creating a new dinner plate, passing in the parameters, and transferring control to the new method is a *method call*.

Once the values are passed in, your parameters behave just like everything else in your memory space.

Figure 4.7 Method Running



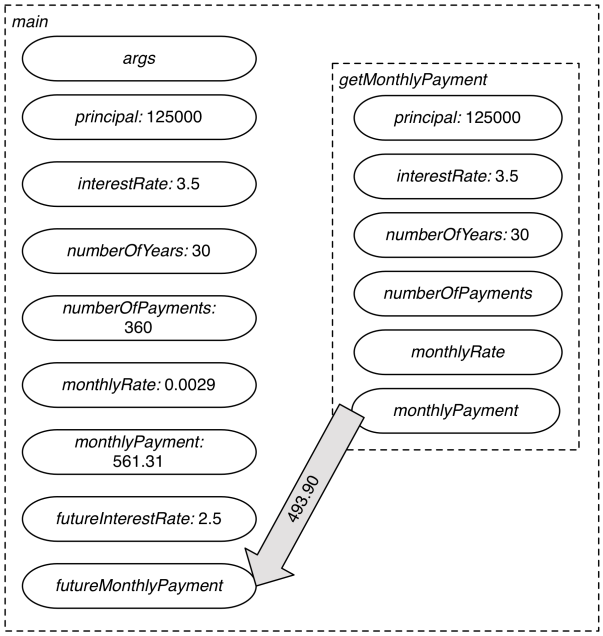
At this point, **`getMonthlyPayment`** is running. Your code in **`main`** will sit there waiting patiently until **`getMonthlyPayment`** finishes its work and returns control to it. So only the method on top of the stack is active.

Now, look at the last line in **`getMonthlyPayment`**:

```
return monthlyPayment;
```

The `return` keyword means, "Drop everything, give this value to the caller, and quit." So when that line of code runs, it will hand over the value in `monthlyPayment`.

Figure 4.8 Returning A Value



With that, the **`getMonthlyPayment(…)`**'s work is done. It will immediately wipe out its dinner plate and everything in it, returning the program back to **`main`**.

DRY: Don't Repeat Yourself

An acronym you see often in programming culture is DRY: Don't Repeat Yourself. It is a little proverb, a chestnut of good advice that all programmers should strive to follow.

Right now, your program repeats this small section of code:

```
int numberOfPayments = numberOfYears * 12;
double monthlyRate = interestRate / 12 / 100; // convert from percentage

double monthlyPayment = principal * (monthlyRate /
        (1 - Math.pow(1 + monthlyRate, -numberOfPayments)));
```

You can find this inside both **`main`** and **`getMonthlyPayment`**. Both bits of code do the exact same thing, and it all works great. So why, if Don't Repeat Yourself is a good idea, is this bad?

People say Don't Repeat Yourself because, if your program is any good, there will come a time when it will be used to do a slightly different job, or in a slightly different situation. Think about what would happen if the place where you lived added a small tax to mortgage payments. You would come back to your program and ask, "Hmm, where did I put the code that makes that calculation?"

If you Don't Repeat Yourself, this job is straightforward. All you do is look through the code for a place where a monthly mortgage payment is calculated. Then you add the code to add the tax.

If you *do* repeat yourself, this job is much harder. After you find the first place, you then have to look over all the rest of the code and make sure that this is the *only* place where you calculate a monthly mortgage payment. You

might find another, and another, and another before you finally satisfy yourself that you have found them all. Once you do that, you then have to add the tax code in every location you found.

In the short run, like right now, this is not a serious problem. In the long run, though, it is a big deal, so it is best to forestall these problems before they occur. Change your code so that it uses the **`getMonthlyPayment`** in *both* places, so that there is only one place the calculation is performed:

Figure 4.9 Unify payment calculation code

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

    int principal = 125000; // in dollars
    int numberOfYears = 30;
    double monthlyPayment = getMonthlyPayment(principal, interestRate, numberOfYears);

    System.out.format("You could pay as little as $%.2f.\n\n", monthlyPayment);

    ...
}
```

The code you will write as you follow this book will strive not to repeat itself in the first place, so this will be the last time we ask you to rewrite code like this. In your own programs, though, you will find that life goes easier if you periodically prune your code to reduce this kind of duplication.

Using IntelliJ To Extract Methods

More often than not, we programmers write code first, and only later decide that we need it to be a standalone method with its own name. When that happens, your IDE can make your job much easier. In this next section, you will use IntelliJ to pull most of **`main`** out into a standalone method that prints out a form letter for one specific person.

The first step is to move a few variables around. Some of the variables you have right now will not be part of your method, but will be its parameters instead. Pull them up to the top to pull them out of your future method:

Figure 4.10 Move parameter inputs out of the way

```
public static void main(String[] args) {
    double interestRate;
    interestRate = 3.5;
    int principal = 125000; // in dollars
    double futureInterestRate = 2.5;

    System.out.println("Greetings!");
    System.out.println();
    System.out.println("If you like low, low mortgage payments,");
    System.out.println("I have an opportunity you will LOVE.");
    System.out.println("Interest rates have fallen to " + interestRate + "%.");

    int numberOfYears = 30;
    double monthlyPayment = getMonthlyPayment(principal, interestRate, numberOfYears);

    System.out.format("You could pay as little as $%.2f.\n\n", monthlyPayment);

    double futureMonthlyPayment =
        getMonthlyPayment(principal, futureInterestRate, numberOfYears);

    ...
}
```

Then add a specific person's name to your message:

Figure 4.11 Add a name to greet

```
public static void main(String[] args) {
    double interestRate;
    interestRate = 3.5;
    int principal = 125000; // in dollars
    double futureInterestRate = 2.5;
    String name = "Billiam";

    System.out.println("Greetings, " + name + "!");
    System.out.println();
    ...
}
```

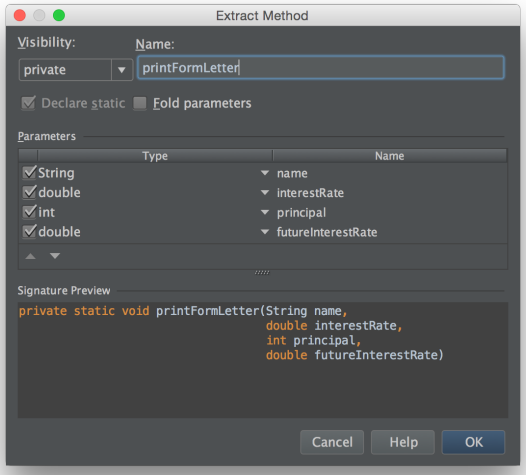
Next, highlight the following section of code:

Figure 4.12 Highlight code to extract into method

```
public static void main(String[] args) {
    double interestRate;
    interestRate = 3.5;
    int principal = 125000; // in dollars
    double futureInterestRate = 2.5;
    String name = "Billiam";

    System.out.println("Greetings, " + name + "!");
    System.out.println();
    System.out.println("If you like low, low mortgage payments,");
    System.out.println("I have an opportunity you will LOVE.");
    System.out.println("Interest rates have fallen to " + interestRate + "%.");

    int numberOfYears = 30;
    double monthlyPayment = getMonthlyPayment(principal, interestRate, numberOfYears);

    System.out.format("You could pay as little as $%.2f.\n\n", monthlyPayment);

    double futureMonthlyPayment =
        getMonthlyPayment(principal, futureInterestRate, numberOfYears);

    System.out.println("Could they fall to " + futureInterestRate + "%?");
    System.out.format("Could you pay $%.2f per month instead?\n", futureMonthlyPayment);
    System.out.println("Of course not! You should refinance immediately.");
}
```

With that section of code highlighted, right click and select Refactor → Extract → Method..., or Cmd-Opt-M on your keyboard. That will pull up a dialog.

Figure 4.13 Extract method dialog



Type in **`printFormLetter`** for the method name. Reorder the `name` parameter to be first in the list as shown above, and select `public` under the Visibility dropdown box. Click OK to extract the method. Your code should then look like this:

Figure 4.14 After method extraction

```
public static void main(String[] args) {
    double interestRate;
    interestRate = 3.5;
    int principal = 125000; // in dollars
    double futureInterestRate = 2.5;
    String name = "Billiam";

    printFormLetter(name, interestRate, principal, futureInterestRate);
}

public static void printFormLetter(String name, double interestRate, int principal,
        double futureInterestRate) {
    ...
}
```

Pretty nifty, eh? Now print out a couple more form letters:

Figure 4.15 Print a couple more form letters

```
public static void main(String[] args) {
    double interestRate;
    interestRate = 3.5;
    int principal = 125000; // in dollars
    double futureInterestRate = 2.5;
    String name = "Billiam";

    printFormLetter(name, interestRate, principal, futureInterestRate);
    printFormLetter("Kristin", interestRate, principal, futureInterestRate);
    printFormLetter("Aaron", interestRate, principal, futureInterestRate);
}
```

Run your program, and you will see three form letters printed out:

```
Greetings, Billiam!
```

```
If you like low, low mortgage payments,
...Of course not! You should refinance immediately.
Greetings, Kristin!

If you like low, low mortgage payments,
...Of course not! You should refinance immediately.
Greetings, Aaron!

If you like low, low mortgage payments,
...Of course not! You should refinance immediately.
```

Challenges

- Add the tax calculation discussed above. Include an additional 5% tax after the initial mortgage calculation.
- Add a new method called **`printFormLetters`** that prints three form letters for one person, each with a different interest rate.
- Add a new method called **`printRandomInterestRate`** that uses the **`Math.random()`** function to print out a form letter with a random interest rate between 2.5 and 4.5. (**`Math.random()`** returns a floating point number between 0 and 1.)

5
Decision Making

A weather app shows a snowy background in winter and a sunny one in summer. A web site checks to see whether your password is correct before letting you in. Every single program you use in the wild makes decisions like this on larger and smaller scales. Each decision changes what the program will do.

There are a few ways you can make decisions in your program, but the most important way is by using *boolean logic*. A boolean value is an answer to a yes/no question, and can be either "yes" (`true`) or "no" (`false`).

In this chapter, you will see how to use boolean logic to get your form letter printer to give the appropriate interest rate for each individual's credit score. You will also use it to validate input: you will make sure that the numbers being sent to your methods will produce outputs that make sense.

Writing Boolean Expressions

Let's jump right in and write a method to decide what your interest rate will be. It will need to know the credit score, so add this as a parameter called `creditScore`. At first, your method will say that until it looks at `creditScore`, the interest rate will be 15%.

Figure 5.1 First pass at interest rate method

```
public class Main {
    ...

    public static double getInterestRate(int creditScore) {
        double interestRate = 15.0;

        return interestRate;
    }
}
```

IntelliJ will make this method gray and put squigglies underneath it because it is not used anywhere. This is not a huge problem. Once you hook it up it will go away:

Figure 5.2 Hook up interest rate method

```
    public static void main(String[] args) {
        int creditScore = 850;
        double interestRate;
        interestRate = getInterestRate(creditScore);
        int principal = 125000; // in dollars

        ...
    }
```

Everyone in the United States has a FICO credit score between 300 and 850. A credit score of 850 is perfect, so it would be nice to give folks with perfect scores a better interest rate.

So is the credit score perfect, or is it not perfect? Write a boolean expression to test whether the credit score is perfect or not.

Figure 5.3 Perfection test

```
    public static double getInterestRate(int creditScore) {
        double interestRate = 15.0;

        boolean isPerfectScore = creditScore == 850;

        return interestRate;
    }
```

`isPerfectScore` will be either `true` or `false`, depending on whether `creditScore` is `850` or not. You use the == operator here to perform an equality test. If the two values are exactly the same, it yields true. (Make sure not to confuse == with =. They look similar, but you will be unhappy if you end up using = where you really wanted ==.)

The right hand side here is an expression, just like the arithmetic expression you used earlier. Boolean operators can be mixed with are evaluated after arithmetic operators. Boolean expressions work just like arithmetic expressions, but use different operators. The equals operator is a boolean operator that is true if the two values it compares are exactly the same. Since `creditScore` is 850 when you call it in **`main`**, `isPerfectScore` should have a value of `true`.

Go ahead and verify this with a print statement:

Figure 5.4 Verify your truthitude

```
    public static double getInterestRate(int creditScore) {
        double interestRate = 15.0;

        boolean isPerfectScore = creditScore == 850;

        System.out.println("Is my credit score perfect? isPerfectScore: " +
            isPerfectScore);

        return interestRate;
    }
```

You should see a confirmation in the first line of your printed output:

```
Is my credit score perfect? isPerfectScore: true
Greetings, Billiam!

If you like low, low mortgage payments,
...
```

Using If Statements To Make Decisions

Boolean values are used in your most basic and common control flow tool, the *if statement*. An `if` statement uses a boolean value to conditionally execute code. They work like your mother's `if` statements: “If you have finished your dinner, then you can have a cookie.” (Moms are probably excellent programmers.)

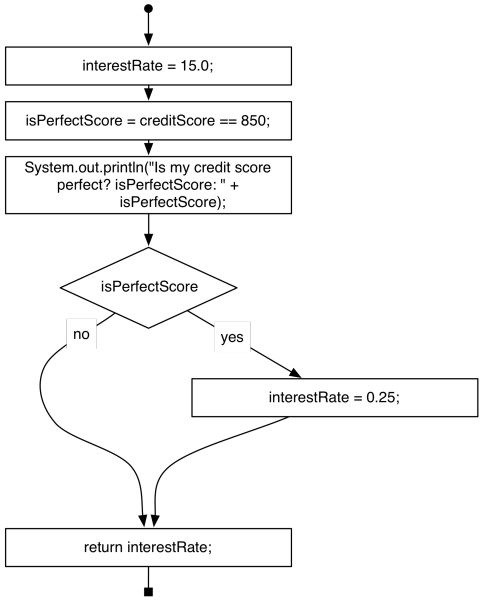
Add an `if` statement to give folks with perfect credit super awesome interest rates:

Figure 5.5 Give folks with perfect credit great interest

```
public static double getInterestRate(int creditScore) {
    double interestRate = 15.0;

    boolean isPerfectScore = creditScore == 850;

    System.out.println("Is my credit score perfect? isPerfectScore: " +
        isPerfectScore);

    if (isPerfectScore) {
        interestRate = 0.25;
    }

    return interestRate;
}
```

Your if statement has two parts: a conditional expression, and a body. The conditional expression goes inside parentheses, and the body goes inside a set of curly braces. If the conditional expression is `true`, then the body is executed. Otherwise, it is skipped.

Figure 5.6 Control flow with an if statement



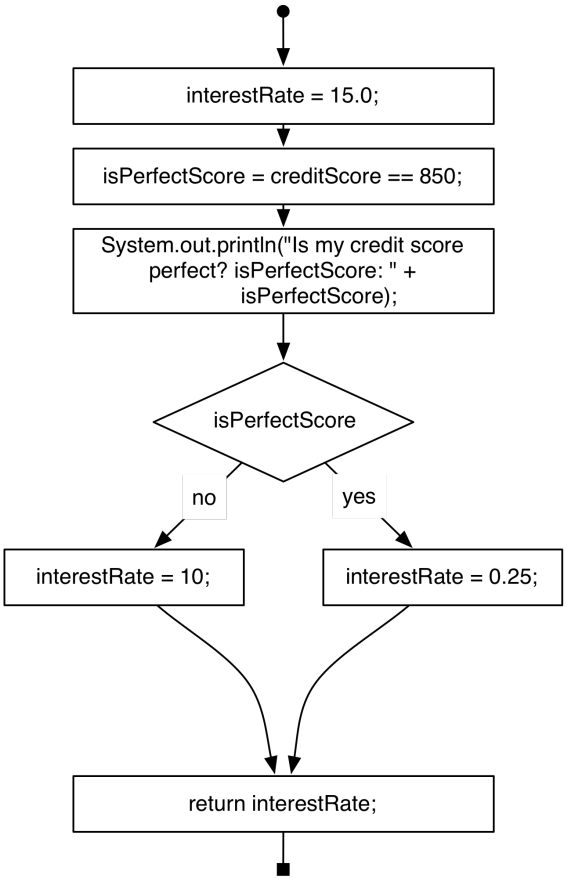
You can also put some code on the left hand side of the fork, by using an *else* statement. Add an else statement to be slightly kinder to the non-perfect souls out there:

Figure 5.7 Add else statement

```
public static double getInterestRate(int creditScore) {
    double interestRate = 15.0;

    boolean isPerfectScore = creditScore == 850;

    System.out.println("Is my credit score perfect? isPerfectScore: " +
        isPerfectScore);

    if (isPerfectScore) {
        interestRate = 0.25;
    } else {
        interestRate = 10;
    }

    return interestRate;
}
```

This would change your control flow diagram to look like this:

Figure 5.8 Using an else statement



Many decisions: If-else chaining

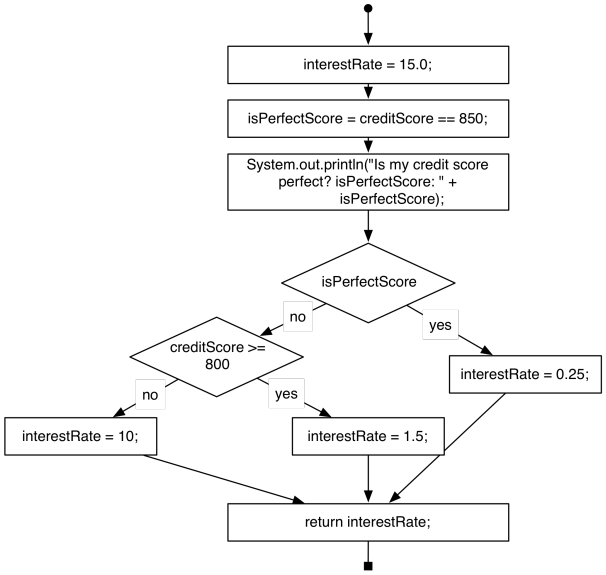
It would be cruel to leave everyone with an 800 or 700 credit score out there in the rain with those unfortunate folks with scores towards the 300 end of things. It would be better if people above 800, but not perfect, received one interest rate, and then folks above 700 received another, slightly worse one, and so on.

You could implement something like that using nested `if` statements:

```
if (isPerfectScore) {
    interestRate = 0.25;
} else {
    if (creditScore >= 800) {
        interestRate = 1.5;
    } else {
        interestRate = 10;
    }
}
```

Which visually would look something like so:

Figure 5.9 Two if statements



This works, but it is not very stylish. Each time you add an if statement, it will walk over to the right a little bit more as you indent each curly brace.

```
if (isPerfectScore) {
    interestRate = 0.25;
} else {
    if (creditScore >= 800) {
        interestRate = 1.5;
    } else {
        if (creditScore >= 700) {
            interestRate = 4.75;
        } else {
            interestRate = 10;
        }
    }
}
```

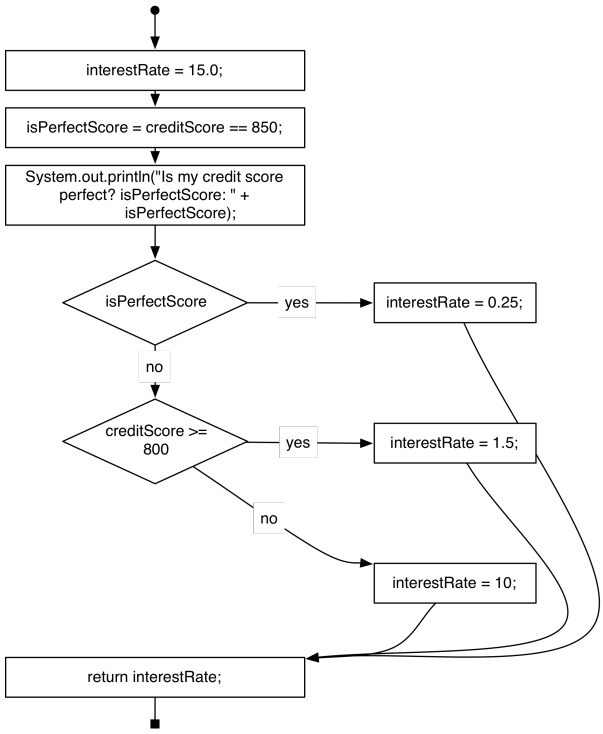
Each time you add more indentation, the code becomes more difficult to visually reason about. There is a fix, though, which is to use an *else if* statement. Write two *else if* statements to add conditions for credit ratings of 700 and 800.

Figure 5.10 Add else if statements

```
public static double getInterestRate(int creditScore) {
    ...

    if (isPerfectScore) {
        interestRate = 0.25;
    } else if (creditScore >= 800) {
        interestRate = 1.5;
    } else if (creditScore >= 700) {
        interestRate = 4.75;
    } else {
        interestRate = 10;
    }

    return interestRate;
}
```

When making an ordered series of decisions, a train of `else if`s is the clearest way to express what your code is deciding to do. Each `if` is checked one after the other. Writing the code this way makes it easier to see just by looking at it that this control flow is still mostly linear:

Figure 5.11 An orderly train of decisions



Combining Booleans: Boolean Logic

Oftentimes, truths are statements about other truths put together. Your mother might say, "If you have eaten all of your dinner *and* you have finished your homework, you can play video games."

The "and" used by your mother here is a common kind of *boolean logic*. You can write boolean logic in your program by using the `&&` (pronounced "and") operator, the `||` (pronounced "or") operator, and the `!` (pronounced "not") operator.

Add one more `if` statement to **`getInterestRate`** that adds a discount of 0.5% for everyone whose score is not perfect, but is also higher than 500.

Figure 5.12 Use some boolean logic

```
public static double getInterestRate(int creditScore) {
    ...
    } else {
        interestRate = 10;
    }

    if (!isPerfectScore && creditScore > 500) {
        interestRate -= 0.5;
    }

    return interestRate;
}
```

You would pronounce the entire first line aloud, “If not is perfect score, and credit score is greater than 500”.

Take a look at the part that says `!isPerfectScore` first. This uses the not operator. The not operator reverses the value of whatever boolean expression immediately follows it. When `isPerfectScore` is `true`, `!isPerfectScore` is `false`. When `isPerfectScore` is `false`, `!isPerfectScore` is `true`.

Next, look at the whole expression: `!isPerfectScore && creditScore > 500`. Each side of the `&&` operator is evaluated first, yielding a boolean value:

```
if (false && true) {
```

The value of the `&&` operator depends on these two values. If both of the values are true, then the entire expression is true. Otherwise, the entire expression is false. Here, since the value on the left is false, the `&&` operator yields false.

The `||` operator is similar, but more generous with the `true`s. The only time `||` yields false is when the operands on either side are both false. Otherwise, it yields true.

Table Of Boolean Operators

There are more boolean operators than you can shake a stick at. So, rather than shaking a stick at them, you can refer to the following table.

Table 5.1 Boolean Operators

Operator	Description
`A && B`	`A` and `B` must both be boolean values. Yields `true` only when `A` and `B` are both `true`. Otherwise, yields `false`.
`A || B`	`A` and `B` must both be boolean values. Yields `false` only when `A` and `B` are both `false`. Otherwise, yields `true`.
`! A`	`A` must be a boolean value. Yields the opposite of `A`. When `A` is `true`, yields `false`. When `A` is `false`, yields `true`.
`A == B`	`A` and `B` may be any type, but must both be the same type. If `A` and `B` have the same value, then `==` is `true`, otherwise it is `false`.
`A != B`	Equivalent to `!(A == B)`.
`A > B`, `A >= B`, `A < B`, `A <= B`	`A` and `B` must both be numeric types. These are the relational operators supported in Java. Note that relational ranges are not supported: `A > B > C` is not a valid expression.

Short-Circuiting

One important detail to know about Java's boolean operators: they do not work exactly like other operators you have used. Take this expression as an example:

```
boolean shouldGoToKitchen = cakeInKitchen && (isHungry() || isSad() || isLonely());
```

If there is cake in the kitchen, and I am hungry, sad, or lonely, I should go to the kitchen and eat that cake. To start evaluating the top-level `&&` expression, the variable on the left is evaluated first.

Listing 5.1 Whether to go to the kitchen, step 1

```
boolean shouldGoToKitchen = cakeInKitchen && (isHungry() || isSad() || isLonely());
```

As it turns out, there is no cake in the kitchen.

Listing 5.2 Whether to go to the kitchen, step 2

```
boolean shouldGoToKitchen = false && (isHungry() || isSad() || isLonely());
```

To finish evaluating the top-level && expression, the right hand side would usually be evaluated next. **`isHungry()`**, **`isSad()`**, and **`isLonely()`** would be called, and if any of those are `true`, the right hand side would be `true`, too.

If you look at the definition of &&, though, you will see that it does not matter whether the right hand side is true or not. If there is no cake in the kitchen, being hungry will make the kitchen no more appealing.

Here is the next step in that evaluation sequence.

Listing 5.3 Whether to go to the kitchen, step 3

```
boolean shouldGoToKitchen = false;
```

Since the right hand side of an && operator does not matter when the left hand side is `false`, the entire expression is immediately evaluated as `false` if the left hand side is false. That means that **`isHungry()`**, **`isSad()`**, and **`isLonely()`** are never called. This is called *short-circuit evaluation*, because it cuts out part of the evaluation process.

Short-circuit evaluation also applies to `||`. For `||`, it is the opposite: if the left hand side is `true`, then the whole `||` must be `true`.

For The More Curious: Syntax Details

If you are a scrupulous kind of person, you like to know exactly how everything works. For those scrupulous people, we provide this bit of language lawyering: `if` statements can work in a few other ways besides the ways we showed you in the code examples.

For an example, take a look at Figure 5.5. This code listing could also be written like this:

```
public static double getInterestRate(int creditScore) {
    double interestRate = 15.0;

    boolean isPerfectScore = creditScore == 850;

    if (isPerfectScore)
        interestRate = 0.25;

    return interestRate;
}
```

See the difference? No braces! Instead, you have a single statement. Out of the box, that is all the `if-else` statement is meant to do: “If this is true, execute this statement. If not, execute this other statement.”

The curly braces let you get around that one-statement limitation. The curly braces you see are actually a shorthand that allows you to squish a whole bunch of statements in the place where one statement would normally go. You can use them anywhere you like, no problem:

```
public static double getInterestRate(int creditScore) {
    double interestRate = 15.0;

    {
        boolean isPerfectScore = creditScore == 850;

        if (isPerfectScore)
            interestRate = 0.25;
    }

    return interestRate;
```

```
}
```

We showed you how to write if statements with braces and sneakily omitted the braceless form for one reason: it is almost universally agreed that the original idea of having if statements without braces was a bad one. The reason is that it is a lot easier to accidentally introduce bugs into the braceless form.

Here is an example: say that you wanted to treat folks with perfect credit scores even better. Not only would you give them an interest rate of 0.25, but you would also print them out a counterfeit $100 bill and mail it to them. So you add some more code to your `if` statement:

```
public static double getInterestRate(int creditScore) {
    double interestRate = 15.0;

    boolean isPerfectScore = creditScore == 850;

    if (isPerfectScore)
        interestRate = 0.25;
        fireUpCounterfeitMoneyPrinter();
        printBillOfDenomination(100);

    return interestRate;
}
```

If you do this, you have a serious problem on your hand. Even though you indented your two new method calls to look like they should go with your if statement, remember: the indentation is only there to make the code easier to read for you. Java does not even realize that the indentation is there. So when Java runs this code, it reads it like this:

```
public static double getInterestRate(int creditScore) {
    double interestRate = 15.0;

    boolean isPerfectScore = creditScore == 850;

    if (isPerfectScore) {
        interestRate = 0.25;
    }

    fireUpCounterfeitMoneyPrinter();
    printBillOfDenomination(100);

    return interestRate;
}
```

Of course, this is not at all what you intended. Now you are printing out a $100 bill for every single person you send out this mail to. You are going to be a very popular, very sad person.

This issue is even dangerous to experienced programmers. Experienced programmers indent their code all the time. Soon enough, they come to rely on code being indented, and do not even see the curly braces. So the mistake made earlier could be missed by anyone, beginner or expert.

There is another older programming language called C, which Java was intended to resemble. C was designed to allow you to use single statements without braces almost everywhere. By the time everyone figured out that this was terrible for everyone, it was too late: everyone had already written too much C that used it, and people were opinionated enough to expect it.

Java tries to fix this where it can. Parts of Java that do not have a C heritage (like exception handling) disallow single statements. Where it borrows from C, though, the original single statement behavior remains the same.

For The More Curious: The Ternary Operator

You can also write a single expression that uses a boolean to evaluate to two different values. Figure 5.5 could also be written in one line like this:

```
public static double getInterestRate(int creditScore) {
```

```
    return creditScore == 850 ? 0.25 : 15.0;
}
```

This complicated line of code accomplishes the exact same thing as Figure 5.5.

You have seen binary operators (operators with two arguments), and a unary operator (`!`, which has one argument). This expression is a *ternary operator*, an operator with three arguments: the conditional expression to the left of the `?`, the value in between the `?` and the `:`, and the value to the right of the `:`. It is the only ternary operator in Java, so if someone says, "ternary operator," this is what they are talking about.

When you evaluate this operator, it first evaluates the part on the left. If it evaluates to `true`, the value on the left of the `:` is evaluated and becomes the value of the whole expression. If the part on the left is `false`, though, the value on the right of the `:` is evaluated.

We will not be using the ternary operator very frequently in this book, nor do we recommend that you use it frequently in your own code. New programmers often believe that they can use this operator to write code like Ernest Hemingway: terse, meaningful, brilliant. In our experience, though, that kind of code usually ends up like a message a CIA agent might write: terse, meaningful, and impossible to read once the agent is dead.

For The More Curious: Recursion

In the previous chapter, you saw how to create and call your own static method. If you are the sort who checked to see if forks fit into electric outlets as a child, you might have wondered, "Hey, can my method call *itself*?"

It turns out you can. Calling yourself is called *recursion*. No special trick is required to do it. Here is what a recursive call to **`getInterestRate`** would look like.

```
public static double getInterestRate(int creditScore) {
    ...

    return getInterestRate(creditScore);
}
```

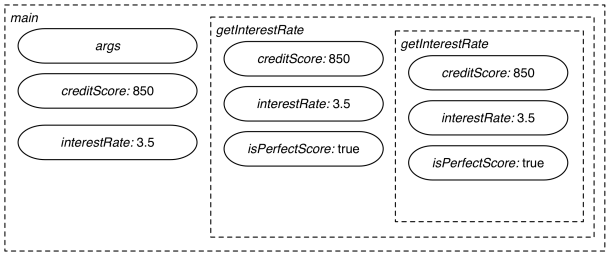
We can tell you what this means in one quick sentence: "Call **`getInterestRate(interestRate)`** and return what it returns." Unless you are familiar with recursion, though, it will require some thought to figure out what that call will do.

Sometimes it is easier to just run a program than it is to think about it. Do that as a first step here to see what happens. Run this code, and you will see a long red splotch of text printed out that begins with the following line:

```
Exception in thread "main" java.lang.StackOverflowError
```

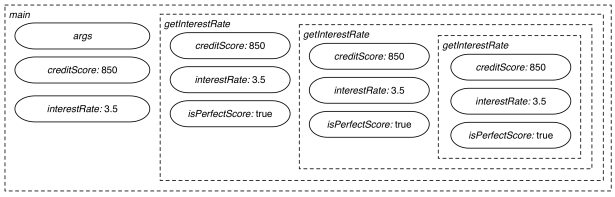
Remember from Figure 4.6 that when a method is called, a new memory area is created for it and placed on top of the old memory area. The same thing happens for a recursive call: **`getInterestRate`** creates an additional space for a second call to **`getInterestRate`** on the stack and hands off control to it:

Figure 5.13 getInterestRate Call #2



And when **`getInterestRate`** call #2 runs, it then calls **`getInterestRate`** *again*, creating yet another space on the stack, which it then hands off control to:

Figure 5.14 getInterestRate Call #3



And when **`getInterestRate`** call #3 runs, it — you get the idea.

This merry-go-round will never stop. Each time it goes around, it will add one more thing to the stack. Eventually, it will run out of stack space. When that happens, you get a **`StackOverflowError`**.

Methods that use recursion must be able to stop this crazy sequence from spiraling out of control. Once one method call returns a value, the whole stack of recursive calls will unwind itself, one by one. Now that you have the `if` statement in your toolbox, you can do that by writing a *termination condition*.

Here is an example of a working recursive method. When you call **`singSong`**, it will print out "99 Bottles Of Beer On The Wall".

```
public static void singSong(int bottles) {
    if (bottles <= 0) {
        System.out.println("No more bottles of beer on the wall, " +
            "no more bottles of beer.");
        System.out.println("Go to the store and buy some more, " +
            "99 bottles of beer on the wall.");
    } else {
        System.out.println("" + bottles + " bottles of beer on the wall, " +
            bottles + " bottles of beer.");
        bottles -= 1;
        System.out.println("Take one down and pass it around, " +
            bottles + " bottles of beer on the wall.");

        singSong(bottles);
    }
}
```

If you call **`singSong(99)`**, this method will run 100 times. The first 99 times, it will skip the first `if` block and go through the `else` block. Run through it in your head to see: `99`, `98`, `97`… each time **`singSong`** calls itself, it will call itself with `bottles` set to one less than it was the previous time through.

This `else` block is called your *iteration condition*. When this branch of code executes, it *iterates*, or moves, to the next thing it should work on.

This method will keep on iterating down, down, down, until `bottles` is `0`. At that point, you have 100 active stack frames, one for each bottle, plus the call to **`singSong(0)`**. When that happens, you hit a brick wall: your termination condition. Your termination condition will *not* make a recursive call.

This brick wall is what it takes to make this recursive method useful. Once that 100th call to **`singSong`** terminates, **`singSong(0)`**'s stack frame can clear out. When that happens, **`singSong(1)`**, which was waiting in its `else` block for **`singSong(0)`** to finish up, can also return. Then **`singSong(2)`**, which was waiting on **`singSong(1)`**, can finish up, which lets **`singSong(3)`** finish up, and so on. Eventually, your call to **`singSong(99)`** will finish up, too.

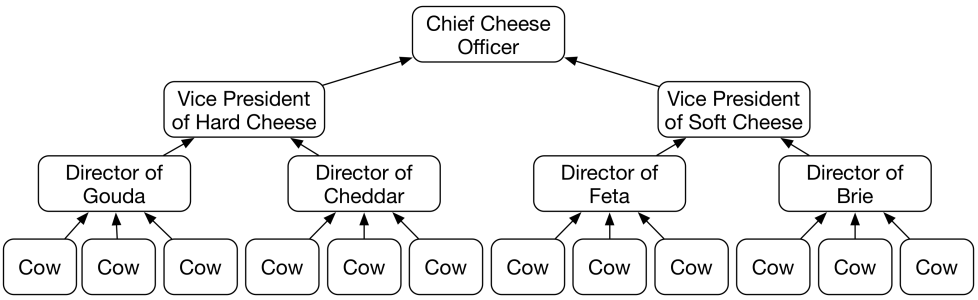
Uses For Recursion

As a way to solve problems, recursion works. As a way to communicate what you mean to someone reading your program, though, it often falls short. Understanding a recursive method like **`singSong`** requires you to think a bit. Thinking is not a bad thing, but stylish programmers prefer to write code that requires you to think only about the important things, not unimportant details.

In the next chapter, you will learn about using `for` loops and `while` loops to iterate. More often than not, those techniques are better ways of iterating than recursion is.

For some problems, though, recursion is a great fit. Take the following question, for example: how many people is an employee at a company responsible for?

Figure 5.15 Making cheese



The hard way would involve a lot of conditional code. “Cows are only responsible for themselves. Directors are responsible for themselves, plus all the cows that report to them. If they are vice president, though, they are responsible for themselves, plus all of their directors, plus all their directors' cows.”

There is a definition that is to the point, though:

1. Cows are only responsible for themselves.
2. Everyone else is responsible for themselves, plus everyone their direct reports are responsible for.

This is a recursive definition. Part #1 is your termination condition, and part #2 is your iteration condition. If you found a way to express this in code, you would have a working implementation. (This will be impossible to do until you learn about objects later in the book.)

For most workaday programmers, understanding and using recursion is a rarely used tool. Having it there and knowing when to use it can be invaluable, though. As in all things programming related, use your best judgement.

6
Switches and Basic Exceptions

In this chapter, you will write a simple calculator that allows you to type in two numbers and an operator, then get the result back. So you will write code to read input from the user's keyboard.

This introduces a new problem: sometimes, reading input does not work correctly. Those characters might be coming from the keyboard, and the keyboard might catch fire. Or they could be coming from across the internet, and the internet could catch fire. The simple truth is that when we say "input", we mean something that comes from the real world, and things often go wrong there. Fires, for example.

In Java, errors are represented and managed using things called *exceptions*. Exceptions are a type of control flow, as well as the first kind of *object* you will work with in this book.

Creating The Calculator Project

Start off by creating another project for your calculator program. Click File → New Project... to create a new project. Click Next, then check the box for Create project from template on the following page. Click Next again, and type in "Calculator" for the project name. Delete the text under Base package, choose where you want your project to go under Project location, and click Finish to create the initial version of your project.

Once you have created your project, open up the **`Main`** class and add in the first part of the app, a method call to fetch a value:

Figure 6.1 Call method to read a number

```
public static void main(String[] args) {
    int firstNumber = readNumber();
}
```

The **`readNumber`** method will not exist, so it will be highlighted dark red in IntelliJ. IntelliJ can automatically fix this for you by creating a new empty method with the appropriate signature. Type Opt-Enter to pull up the autofix menu, which will display one option: Create Method 'readNumber'. Select it, and it will autocomplete the following code for you:

Figure 6.2 Initial readNumber implementation

```
public static int readNumber() {
    return 0;
}
```

Reading Bytes From Stdin

Receiving input from the keyboard is done through `System.in`, the complement to `System.out`, which you have been using to print things out. `System.in` reads in data one byte a time. For the kinds of characters you will be working with in Calculator, each byte represents a single character typed in by the user. This byte-at-a-time way

of doing things is much smaller scale than the way you worked with `System.out`. There are higher-level ways of working with `System.in`, but they require the use of objects, which we will get into later on in this book.

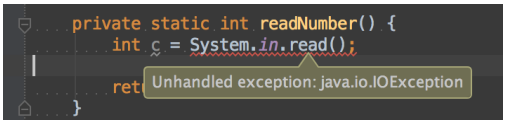
To read in a single byte, call the **`read`** method on `System.in`.

Figure 6.3 Call read method

```
public static int readNumber() {
    int c = System.in.read();

    return c;
}
```

When you type this in, the call to **`read`** will be highlighted in red. Hover your cursor over the call to **`read`** briefly, and you will see that the problem is with an unhandled exception.

Figure 6.4 I say hmm, what's going on?



`System.in.read()` throws a type of exception called a **`java.io.IOException`**. To fix this, you will need to know what an exception is, why you should care about whether it is handled or not, and how you would go about making this one handled.

Exceptions

An exception is something that happens in a method that is outside of the way that it is intended to work. A good example of an exception would be on a method that returns the text of a web page pulled across a network: normally, the web page comes back, and everything is fine. If the internet is down, though, you will receive an error instead. In Java, this would be an exception.

The exception is both an *event* that happens during your program, and an *object* that represents that event. So you could refer to the event that happened by saying, "While calling **`System.in.read()`**, an exception occurred." You could also refer to the object itself by saying "An exception was thrown," or "An exception was raised."

To fix this uncaught exception, you *catch* the exception by writing a `try-catch` block.

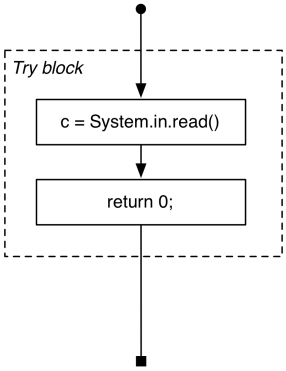
Figure 6.5 Catch your first exception

```
public static int readNumber() {
    try {
        int c = System.in.read();

        return c;
    } catch (IOException exception) {
        System.out.println("We tried to read a number, " +
                "but this exception occurred: " +
                exception);

        return 0;
    }
}
```

With that, your errors will be cleared away.

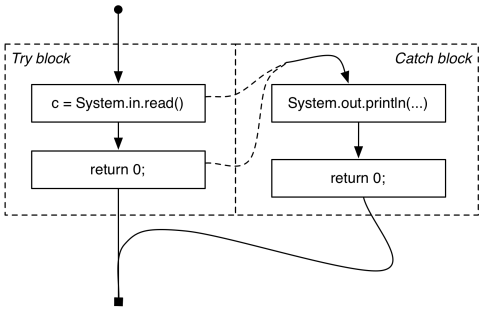
Now to take a look at what your new code does. The first step in catching an exception is to say where the exception could be coming from. You do this by wrapping a section of code inside a *try block*.

Figure 6.6 A try block



When an exception is thrown, the code that throws the exception immediately stops what it is doing and jumps to some other code that tries to take corrective action. The `try` block you define defines a section of code that you want to handle exceptions for. If an exception occurs anywhere inside the `try` block, control flow will immediately jump to the *catch block* that comes right after your try block, where you take your corrective action.

Figure 6.7 Try-catch



Think of the `try` block as your Plan A. If Plan A comes off without a hitch, then that is the only plan you need. If it does not, though, you go immediately jump to your backup, Plan B, no matter where in Plan A you are.

When you have to switch to Plan B, you want to know why Plan A went wrong in the first place. This is why your `catch` block has a named, typed variable inside parentheses: to stash the exception that occurred. This exception object contains information about the exception. This also defines what kind of exception you are catching: if something besides an **`IOException`** were thrown here, your catch block would ignore it.

In this chapter, the code to handle the **`IOException`** is purely there as a precautionary measure. **`IOException`** is a *checked exception*, which means that you are *required* to catch it somewhere if a method says that it throws it. For regular keyboard input, though, no exception will ever be thrown. It is there in case your input comes from an unreliable source, like a network connection.

Characters vs. Numbers: Representation Of Data

In **`readNumber`**, you call **`System.in.read`**, which returns an `int`, which is a number. A number is not always only a number, though. Often, it is something more, something else.

Think about the latin alphabet for the moment. As a thinking aid, we present you with: the latin alphabet, plus its friends, the digits 0-9.

```
A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W

X  Y  Z  0  1  2  3  4  5  6  7  8  9
```

Each one of these letters and digits is called a *character*. A computer cannot store characters, though: it can only store numbers. So to store characters, programs assign a number to each character:

```
A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R  S  T  U  V  W
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19 20 21 22

X  Y  Z  0  1  2  3  4  5  6  7  8  9
23 24 25 26 27 28 29 30 31 32 33 34 35
```

That is what your integer is: a number representing a character. All you need to do is find which number character it represents. Which you can do by calling a static method:

Figure 6.8 Getting a number from an encoded character

```
public static int readNumber() {
    try {
        int c = System.in.read();

        return Character.getNumericValue(c);
    } catch (IOException exception) {
        System.out.println("We tried to read a number, " +
                "but this exception occurred: " +
                exception);

        return 0;
    }
}
```

The variable `c` is a character's number. If that character is a digit, **`Character.getNumericValue`** returns the number for that digit. Otherwise, it returns `-1`, which means that the character is not a digit at all.

Add a little bit of code to **`main`**, and you should be able to see this working a little bit. Print out the number that was input:

Figure 6.9 Print first number

```
public static void main(String[] arguments) {
    int firstNumber = readNumber();

    System.out.println("First number: " + firstNumber);
}
```

Run your app, and initially you will see nothing in the Run window. This is because it is waiting (called “blocking” in programming jargon) for you to type something in. Click inside the window, type in a number, and hit enter. You should see your calculator print out the same number you typed in:

```
5
First number: 5
```

By default, Java must wait until you hit the Enter key before it sees anything that you type. It is possible to change this behavior so that it sees each character as you type it, but doing so is an involved process that changes depending on what kind of operating system you run. So, sadly, we must omit it.

Oh yeah: remember, this program reads in only a single character, not a number. So if you type in `10`:

```
10
First number: 1
```

You will see `1` printed out.

Declaring Checked Exceptions

The complete program will take in input that looks like this:

```
5+6
```

So the next step is to read in the rest of the input: the operator, and the second number. Write out the code to do that:

Figure 6.10 Read in remaining input

```
public static void main(String[] arguments) {
    int firstNumber = readNumber();

    int operator = readOperator();

    int secondNumber = readNumber();

    System.out.println("First number: " + firstNumber);
}
```

The **readOperator** method does not exist yet, so put your cursor on it and type Opt-Enter to pull up your autofix selections. Select Create Method 'readOperator' to create a new **readOperator** method stub, and fill it out like so:

Figure 6.11 Create readOperator method

```
public static int readOperator() {
    int c = System.in.read();

    return c;
}
```

Now you have the same problem you had in **readNumber**: **System.in.read** throws an exception. When you were reading in one number, catching that exception immediately made as much sense as anything else. Now that you want to read in two numbers and an operator, catching it immediately means that you will be catching the same kind of exception three times.

This may be what you want, but you might rather give instructions like this: “Go read in a number, an operator, and another number. If anything goes wrong during all of that, print out what went wrong, say ‘I give up!’ and quit.” To do that you need to declare your own checked exceptions.

Let's use checked exceptions to centralize your error handling. Start by adding a *throws clause* to **readOperator**:

Figure 6.12 Add throws clause to readOperator

```
public static int readOperator() throws IOException {
    int c = System.in.read();

    return c;
}
```

When you do this, the red error line underneath **System.in.read** should disappear. Another one will appear underneath your call to **readOperator** in **main**. Ignore the new error in **main** for now, and move back to **readNumber**. Get rid of its old try-catch and add a `throws` there, too:

Figure 6.13 Add throws clause to readNumber

```
public static int readNumber() throws IOException {
    int c = System.in.read();

    return Character.getNumericValue(c);
}
```

Return your attention to **main**. You will now see error marks underneath your calls to both **readOperator** and **readNumber**. Adding a `throws` clause to your **readNumber()** implementation made it declare a checked exception, exactly like **System.in.read()**. So this is the exact same error.

A `throws` clause is a declaration of intent. It means that you reserve the right to throw the kind of exception named in the throws clause. Since you reserve that right, you can call methods like **System.in.read()** which throw the same kind of exception without handling them. It is a way of passing the buck to the person calling your method.

So while you have got rid of one `try-catch`, you still need to have one somewhere to catch the exceptions now thrown by **`readNumber`** and **`readOperator`**. Do that by adding a `try-catch` to **`main`**. IntelliJ can make this a little easier for you. First, select the highlighted code:

Figure 6.14 Add a try-catch to main

```
public static void main(String[] arguments) {
    int firstNumber = readNumber();

    int operator = readOperator();

    int secondNumber = readNumber();

    System.out.println("First number: " + firstNumber);
}
```

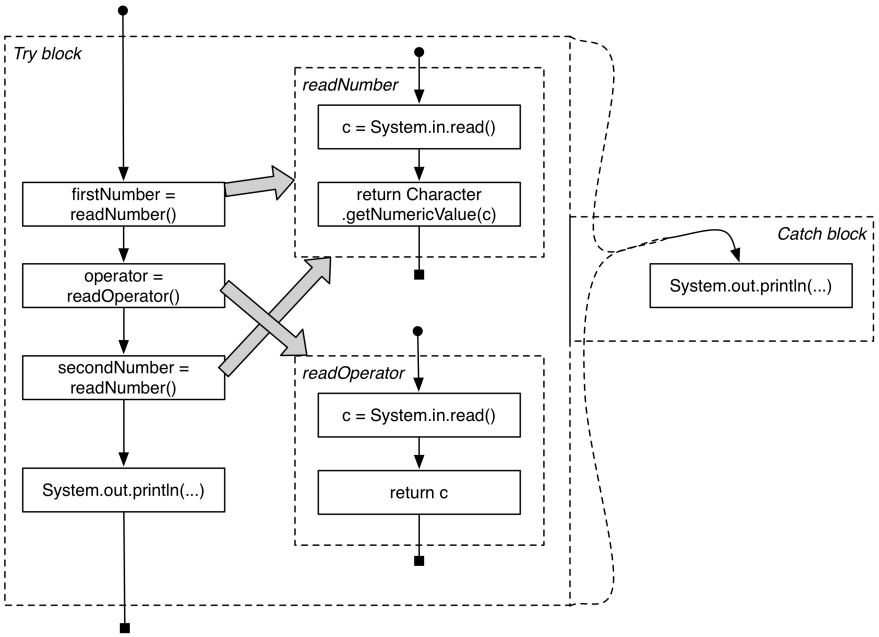
Then select Code → Surround With... (Opt-Cmd-T). In the popup that appears, select try / catch. It will automatically fill out your `try-catch` with some boilerplate code. Fill it in like this:

Figure 6.15 Finish adding try-catch

```
public static void main(String[] arguments) {
    try {
        int firstNumber = readNumber();

        int operator = readOperator();

        int secondNumber = readNumber();

        System.out.println("First number: " + firstNumber);
    } catch (IOException exception) {
        System.out.println("I give up! " + exception);
    }
}
```

Now you have all your error handling in one place. Your control flow now looks like this:

Figure 6.16 More exception control flow



Any one of these statements in `try` could jump to `catch` if they wanted to.

Casting

Add in a bit more code to print out the operator and the second number:

Figure 6.17 Print out two more items

```
public static void main(String[] arguments) {
    try {
        ...

        System.out.println("First number: " + firstNumber);
        System.out.println("Operator: " + operator);
        System.out.println("Second number: " + secondNumber);
    } catch (IOException e) {
        System.out.println("Could not read the equation you typed in: " + e);
    }
}
```

This code will print out the first number, the second number, and... the code point for the operator. You know all about code points, of course, but your users are going to be very confused by the idea that their operator is actually a number.

What you want is not the code point, but the character represented by the code point. In this case, you can get the character by *casting* the `int` value of `operator` to a `char`. A `char` holds a 16-bit integer, just like a `short` does, but Java knows that the number a `char` holds really represents a character. (A UTF-16 encoded character, if we want to be pedantic about it. Google for the documentation of the Java **`Character`** class if you want to dive even deeper.)

To cast a value, prefix it with the type you want to cast it to inside parentheses:

Figure 6.18 Cast to char

```
public static void main(String[] arguments) {
    try {
        ...

        System.out.println("First number: " + firstNumber);
        System.out.println("Operator: " + (char)operator);
        System.out.println("Second number: " + secondNumber);
    } catch (IOException e) {
        System.out.println("Could not read the equation you typed in: " + e);
    }
}
```

The expression `(char)operator` means, "Take this 32-bit integer value and stuff it inside a 16-bit value, then treat that 16-bit number like a character." A 32-bit value holds more information than a 16-bit value, so some of those bits will be thrown away. The bits furthest to the left of the decimal point are thrown away first. In this case, **`System.in.read`** only returns a 16-bit values when a valid character is read, so you are not in danger of losing important information.

Run your program, and you should be able to see both input numbers and the operator you typed in:

```
1+1
First number: 1
Operator: +
Second number: 1
```

Switch Blocks

The next step for your calculator is to perform the operation the user types in. You can accomplish this with an `if-then` tree, but this particular task is better expressed with a *switch block*. A `switch` allows you to say, "Look at this thing for me. If it has this value, do one thing, but if it has this other value, do this other thing." A `switch` can oftentimes be more clear than an `if-else` true for logic like this.

Write a switch block that performs the calculation the user typed in.

Figure 6.19 Write switch statement

```
public static void main(String[] arguments) {
    try {
        ...
        System.out.println("Second number: " + secondNumber);

        double result = 0;

        switch (operator) {
            case '+':
                result = firstNumber + secondNumber;
                break;
            case '-':
                result = firstNumber - secondNumber;
                break;
            case '*':
                result = firstNumber * secondNumber;
                break;
            case '/':
                result = (double)firstNumber / secondNumber;
                break;
            default:
                System.out.println("Unknown operator: " + (char)operator);
                break;
        }
    } catch (IOException exception) {
        System.out.println("I give up! " + exception);
    }
}
```

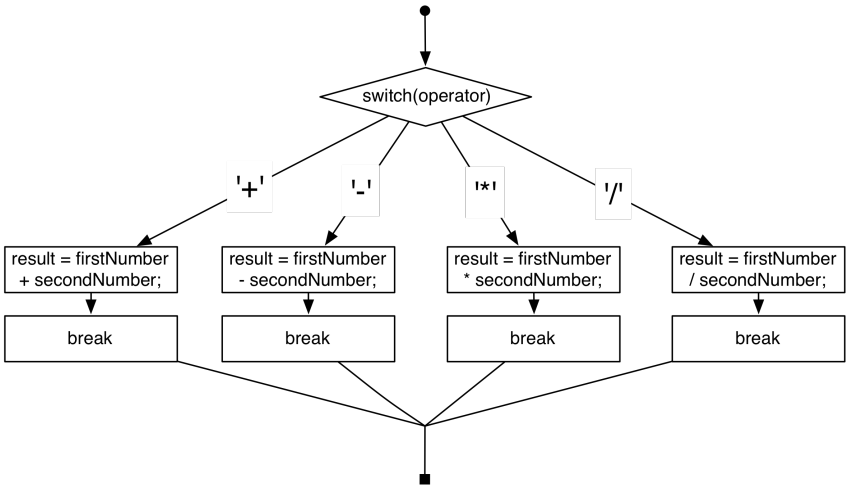
A `switch` block has two parts: a switch value, and a body. The body contains a set of *case labels*. Each `case` label has a possible value that switch value would be equal to. If the switch value is equal to the value in the `case` label, control flow will jump to that `case` label. So if `operator` is equal to the character literal `'-'`, your program will start executing the statement `result = firstNumber - secondNumber`.

The `break` statement that you see all over the place is how you get *out* of the `switch` block. When you hit a `break`, control flow goes right to the end of the block.

At the end, you see a label called `default`. If none of the other `case` labels match the switch value, control jumps to the `default` label.

So for the entire block, control flow looks like this:

Figure 6.20 Switch Block Control Flow



Now that you have your result, print it out:

Figure 6.21 Print out your result

```
public static void main(String[] arguments) {
    try {
        ...

        double result = 0;

        switch (operator) {
            ...
        }

        System.out.println("Result: " + result);
    } catch (IOException exception) {
        System.out.println("I give up! " + exception);
    }
}
```

Try running your app one more time, and you will have a working calculator:

```
1+1
First number: 1
Operator: +
Second number: 1
Result: 2.0
```

Throwing Exceptions

All this talk about exceptions would hardly be any fun without creating them. Here, you have an opportunity to use one: the user can type in an operator that your program does not know about. If that happens, there is not much you can do apart from throwing up your hands and saying, “I have no idea what 5 $ 2 is.” Your program cannot throw up its hands, but it can throw an exception.

Add some code to **`readOperator`** to do a little input validation. You can use a `switch` block again to do the check. It is possible to group multiple case labels together for the same section of code, so use that to make your job a little easier:

Figure 6.22 Check operator type

```
public static int readOperator() throws IOException {
    int c = System.in.read();

    switch (c) {
        case '+':
        case '-':
        case '*':
        case '/':
            break; // If c is one of these, we are fine
        default:
    }
    return c;
}
```

If the `default` label is reached here, you know you received an unsupported operator type. This is where you want to throw an exception:

Figure 6.23 Throw an exception

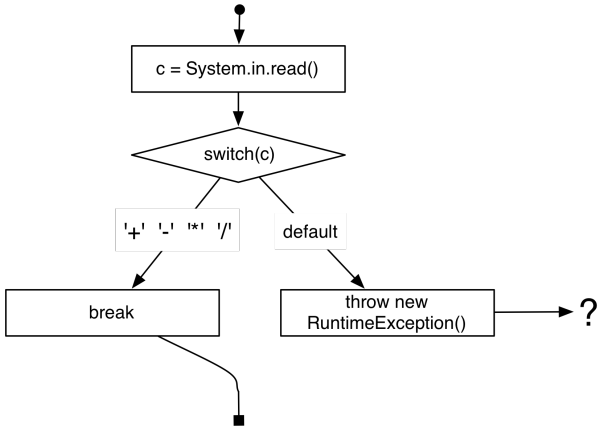
```
public static int readOperator() throws IOException {
    int c = System.in.read();

    switch (c) {
        case '+':
        case '-':
        case '*':
        case '/':
            break; // If c is one of these, we are fine
        default:
            throw new RuntimeException("Unknown operator: " + (char)c);
    }
    return c;
}
```

To throw an exception, you use the `new` keyword to create an exception object and use the `throw` keyword to throw the exception. You will learn all there is to know about about `new` and objects later on in Chapter 9. The `throws` keyword, on the other hand, causes control flow to immediately jump to whoever is setup to catch the exception.

The exception you created is a **`RuntimeException`**. This is a special kind of exception called an *unchecked exception*. You can throw unchecked exceptions without declaring them in a `throws` clause.

If you look at the control flow for the method itself, it looks like this:

Figure 6.24 Control Flow When A Method Throws An Exception

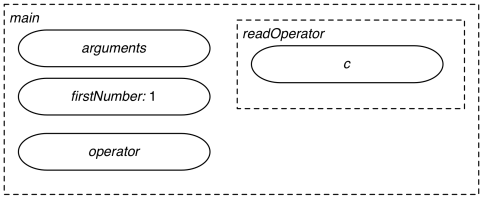


If the character is one you expect, the control flows occurs like you would normally expect it to.

If it is not, though, and your code goes to the `default` label, things get funky. Your code executes the `throw` statement, at which point an exception is triggered. When that happens, your method will immediately exit. It will not return any value. Instead, it will jump to the nearest matching `catch` block in the stack.

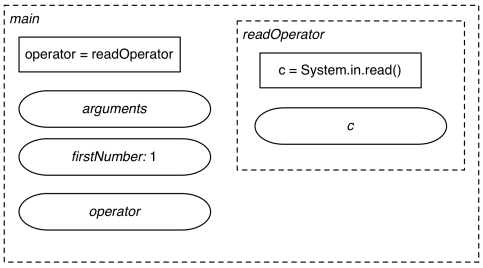
Remember the stack? You were worried about it more when we were discussing the contents of stack frames, but it is still there. When your program runs and hits this `throw` statement, here is what the stack data will look like:

Figure 6.25 Stack Data



In addition to data, your stack contains one more thing: where your method is in execution. So in addition to a bunch of data, your dinner plate has a pointer to a specific statement in your method's control flow.

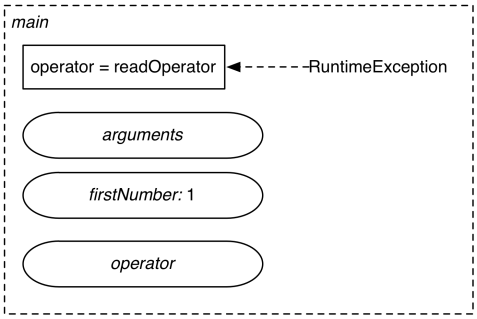
Figure 6.26 Stack Trace



The information about which line of execution each frame is on is called a *stack trace*, because it acts like a trail of bread crumbs. You can use it to track a deeply nested execution back to its origin.

Let's look at the stack trace at the moment you run the statement `throw new RuntimeException()`. When that happens, your method will immediately die and throw the exception.

Figure 6.27 Throwing An Exception, Part 1



Once the exception is thrown, it is going to look for something to catch it. So inside of **`main`**, it will look to see if it is inside a `try` block with a matching `catch` for the right kind of exception.

If you look at the line `operator = readOperator()` in **`main`**, though, you will see that while it is inside a `try` block, the only `catch` block has a type of **`IOException`**, which is not the same as **`RuntimeException`**. Since there is nothing to catch the exception, **`main`** does the same thing that **`readOperator`** did: it immediately dies and throws the exception.

Figure 6.28 Throwing An Exception, Part 2

Of course, once main dies and throws the exception, there are no other stack frames left. So what happens? Trigger the code to throw your exception to see:

```
5$2
Exception in thread "main" java.lang.RuntimeException: Unknown operator: $
    at Main.readOperator(Main.java:49)
    at Main.main(Main.java:8) <5 internal calls>
```

Java catches the exception for you, prints it out, and prints out the stack trace so that you know where it happened. This is called an uncaught or unhandled exception. The printed stack trace uses line numbers to refer to statements, so it has the same information as the rectangular boxes in Figure 6.26.

When an unhandled exception occurs, the entire program will immediately stop, no matter what is happening. Anytime a program halts abruptly like that, it is called a *crash*. Crashes are always bad, but the stack trace will usually at least tell you where the problem is. If an unexpected problem is detected in a program, crashing is often the best thing to do.

Multiple Catch Blocks

The last thing is to catch the **`RuntimeException`**. You could do that by wrapping **`main`** in yet another `try-catch`, but it is simpler and clearer to add another `catch` block. Add the following code to **`main`** to handle an incorrect operator more gently than before:

Figure 6.29 Catch RuntimeException

```
public static void main(String[] arguments) {
    try {
        ...
    } catch (IOException exception) {
        System.out.println("I give up! " + exception);
    } catch (RuntimeException exception) {
        System.out.println("" + exception);
    }
}
```

Now you should see more pleasant output for your user:

```
5$2
java.lang.RuntimeException: Unknown operator: $
```

For The More Curious: Fall through

Almost every switch statement you write as a programmer will have a `break` at the end of every label. It is possible to leave them off, though:

```
double result = 0;

switch (operator) {
    case '+':
        result = firstNumber + secondNumber;
    case '-':
        result = firstNumber - secondNumber;
    case '*':
        result = firstNumber * secondNumber;
    case '/':
        result = (double)firstNumber / secondNumber;
    default:
        System.out.println("Unknown operator: " + (char)operator);
}
```

To see what happens, mentally step through the code above with a value of `'+'` for `operator`. Control would first move to the line `result = firstNumber + secondNumber`, where it would assign an add result to `result`.

When control moves to the next line, `case '-':`, nothing happens at all. Your program simply moves on to the next line after the `case` label. That is because `case` labels are only labels for places in code. They do not do anything.

That means that with `operator` equal to `'+'`, your program would run the following sequence of statements:

```
result = firstNumber + secondNumber;
result = firstNumber - secondNumber;
result = firstNumber * secondNumber;
```

```
result = (double)firstNumber / secondNumber;
System.out.println("Unknown operator: " + (char)operator);
```

This behavior is called *fall through*, because control falls through code for `case` labels your switch value did not match. Usually this is a bug, and the programmer accidentally left out the `break` statements. It is possible to intentionally use this behavior, but it is frowned upon because such code is more difficult to understand.

While relying on fall through is usually a bad idea, it is good practice to use multiple `case` labels for the same section of code. For example, if you wrote the following code, your users could use both `/` and `%` to divide:

```
double result = 0;

switch (operator) {
    case '+':
        result = firstNumber + secondNumber;
        break;
    case '-':
        result = firstNumber - secondNumber;
        break;
    case '*':
        result = firstNumber * secondNumber;
        break;
    case '/': case '%':
        result = (double)firstNumber / secondNumber;
        break;
    default:
        System.out.println("Unknown operator: " + (char)operator);
        break;
}
```

7
Iteration (a.k.a. Repetition)

You calculator program is stubborn. It performs one operation, then quits. Between that and a sophisticated program like a web browser, there is a lot of ground to cover.

One of the single most important waypoints is the idea of repetition, of repeating a step until you are ready to move on. The programming word for this idea is iteration. Iteration is important when you need to repeat an operation many times, but it is also important if your app needs to wait for an event in the real world.

In this chapter, you will use iteration to add the ability to read in numbers with any number of digits. You will then augment your calculator so that it can wait patiently for you to type in equations.

For Loops

Your first job is to use iteration to read in multiple digit numbers. Initially, you will read in a fixed number of digits. Once that is done, you will see how to vary the number of digits.

You can already read in multiple digits if you like, of course. All you need to do is call **`readNumber`** two, three, or more times and then put them all together. Iteration can make this idea a lot easier to write in code, though.

Most iteration in Java is done by using one of a few kinds of blocks called *loops*. (When you see what the control flow looks like, you will see where the name "loop" comes from.) Your first kind of loop is called a *for loop*.

Inside Calculator, write a `for` loop to read in four characters:

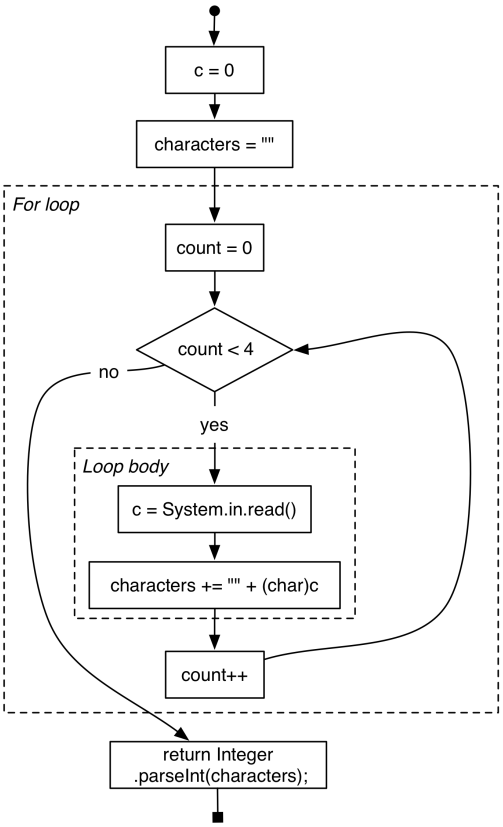
Figure 7.1 Writing a `for` loop

```
public static int readNumber() throws IOException {
    int c = 0;
    String characters = "";

    for (int count = 0; count < 4; count++) {
        c = System.in.read();
        characters += "" + (char)c;
    }

    return Integer.parseInt(characters);
}
```

In English, this loop means, "First, set count to 0. Then, as long as count is less than four, read in a character and add it to your characters string. Each time you repeat, add one to count."

Here is what the control flow for your `for` loop looks like:

Figure 7.2 `For` loop control flow



A `for` loop allows you to repeat a body of statements inside a set of curly braces a few times. The repetition of the loop is usually controlled by a counter that keeps track of the iteration. The `for` loop lets you write the code for the counter inside three statements stuck together inside the parentheses:

1. The first statement, `int count = 0`, is called the *initializer*. "Before you start running the loop, do this." Any variables declared in the initializer will be available only inside the loop.

2. The second statement, `count < 4`, is called the *termination condition*. The termination condition is a boolean expression that is evaluated each time before entering the loop body. It works like an `if` statement: if its value is `true`, the loop body is run. If it is `false`, your code will exit the loop.

3. The final statement, `count++`, is called the *increment*. This is where you move on to the next thing you want to do. You use the increment statement to say, "Okay, I just ran the loop one more time."

Your increment statement uses an operator you have not seen before: ++. The ++, or *increment operator*, adds one to a variable, and the *decrement operator*, --, subtracts one.

So the final effect is to perform the following steps:

1. Initialize `count` to `0`.

2. Run the body of the loop 4 times. Each time the body of the loop is run, `count`'s value is incremented by one, and `count < 4` evaluates to `true`.

3. After the 4th time the loop is run, `count++` has been run 4 times, so `count` is equal to 4. This time, the `count < 4` check is performed and evaluates to `false`.

Run your program, and it will now work with 4 digit long numbers:

```
1234+1234
First number: 1234
Operator: +
Second number: 1234
Result: 2468.0
```

Breaking Out Of Loops

Your next step is to make your calculator program work with numbers with any valid number of digits. To do this you will need to know when the number ends. One straightforward way to do this is to use a specific character to mean, "All right, computer, this number is over." If you use a newline to do that, then your input will look like this:

```
5220
+
91
```

Once you see that newline, you will want your program to say, "Okay, stop this number-reading loop. I want off!"

Write the following code in **`readNumber`** to do this:

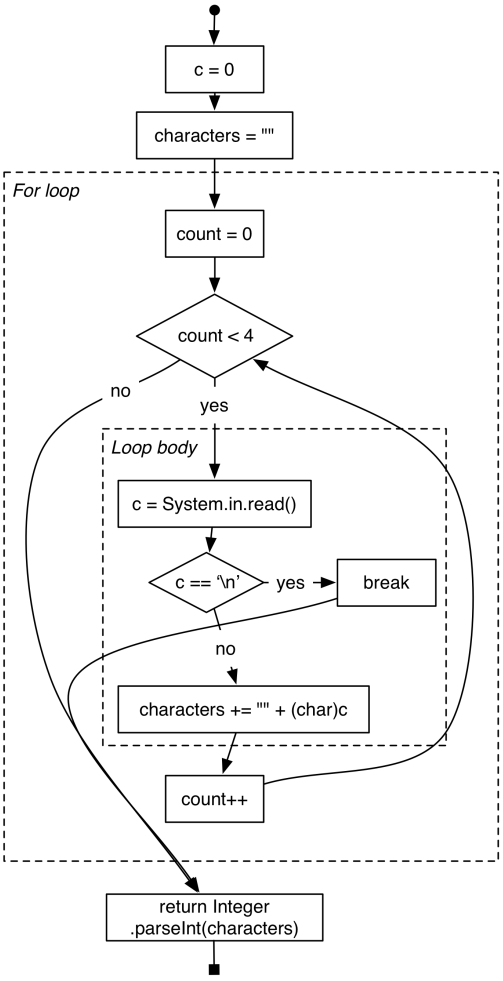
Figure 7.3 Breaking out

```
public static int readNumber() throws IOException {
    int c = 0;
    String characters = "";

    for (int count = 0; count < 4; count++) {
        c = System.in.read();

        if (c == '\n') {
            break;
        }

        characters += "" + (char)c;
    }

    return Integer.parseInt(characters);
}
```

Inside your `if` statement, you have written a new statement called `break`. The `break` statement means, "Break me out of the loop I am in right now." If you have one loop nested inside of another, `break` will only get you out of the innermost loop.

So where before there was only one way to get out of your loop (the termination condition), now there are two:

Figure 7.4 Control flow with break



Add code to your **`readOperator()`** implementation that adds a newline terminator to it, too.

Listing 7.1 Add terminator to readOperator

```
private static int readOperator() throws IOException {
    int c = System.in.read();

    switch (c) {
        ...
    }

    int separator = System.in.read();
    if (separator != '\n') {
        throw new RuntimeException("Unexpected character: " + (char)separator);
    }

    return c;
}
```

Try running your program with new input, making sure to include newlines after each input.

```
123
+
1234
First number: 123
```

```
Operator: +
Second number: 1234
Result: 1357.0
```

Defining constants

Now that you can parse longer numbers, you can safely bump up the maximum length your numbers can be. You will want to make them longer, but not so long that your program will not work.

That length will be an important number to anyone reading your program. Oftentimes you find that a specific value like that is important in your code. You could write the value right in your code (like you did in ???), but then nobody will have any idea what the significance of it is. It will be an unexplained magic incantation. (Hence the term of art: *magic numbers*.)

If you give these values names inside your code, though, whoever reads it can see what special meaning it has. We call these values *constants*, because they never change.

Java defines many useful constants for you. For example, there is a constant value for the largest number you can put in a Java `int`, called `Integer.MAX_VALUE`. You can check its value for yourself if you like, but (spoiler alert) here it is: `2147483647`, which is ten digits long. You will want your numbers to have *less* than ten digits, so that they cannot ever be greater than this value.

Now to add your own constant. Add a constant to **`Main`** called `MAX_DIGITS`:

Figure 7.5 Add constant value

```
public class Main {

    public static final int MAX_DIGITS = 9;

    ...
}
```

The constant is like a variable you define outside of any method or object that anyone can see, and whose value does not change. Each one of those qualifiers before the name `MAX_DIGITS` communicates one part of that requirement:

- `public` means that anyone can see it
- `static` means that it is not attached to any object (like the static methods you have been defining)
- `final` means that `MAX_DIGITS` will never change. (You might say that it has received its *final* assignment.)

By convention, Java programmers write constants in all caps, with underscores separating each word.

Now that you have your constant, use it in your code. Your loop will need to run `MAX_DIGITS + 1` times: one time for each digit, and then once for the space. Write code to do this:

Figure 7.6 Check for maximum digits

```
public static int readNumber() throws IOException {
    int c = 0;
    String characters = "";

    int count;
    for (count = 0; count < MAX_DIGITS + 1; count++) {
        ...
    }

    if (c != '\n') {
        throw new RuntimeException("Too many digits: " + characters);
    }

    return Integer.parseInt(characters);
}
```

You need to include one more last check after the loop is over, to verify that your loop exited by a newline. This will happen when the user fails to include a final newline character, which can only happen when they have entered too many digits.

Remember that variables defined inside the initializer of your `for` loop are only visible inside the body of the loop. Defining `count` before the loop starts here lets you check how many times the loop ran after it completes.

Using Iteration For Interactivity

Your last step in this chapter will be to make your program *interactive* — instead of giving your program an equation and looking at the result, you will have a back-and-forth conversation with it. After reading in an equation, it will print out the result, and then wait for another equation.

The first step in doing this will be to clean your plate: take what you are currently doing in **`main`**, and give it its own name, **`readEquation`**. Highlight everything inside your `try` block:

Figure 7.7 Extracting readEquation

```
public static void main(String[] arguments){
    try {
        int firstNumber = readNumber();

        ...

        System.out.println("Result: " + result);
    } catch (IOException exception) {
        System.out.println("I give up! " + exception);
    } catch (RuntimeException exception) {
        System.out.println("" + exception);
    }
}
```

And extract a method called **`readEquation`** with the Cmd-Opt-M keyboard shortcut.

Figure 7.8 After extracting readEquation

```
public static void main(String[] arguments){
    try {
        readEquation();
    } catch (IOException exception) {
        System.out.println("I give up! " + exception);
    } catch (RuntimeException exception) {
        System.out.println("" + exception);
    }
}

public static void readEquation() throws IOException {
    ...
}
```

While Loops

Most interactive programs do not loop in the way you did with the `for` loop earlier. That is, they do not have some fixed number of times they need to loop. Instead, they loop as long as they need to. Maybe forever! (Probably not — most users will eventually die.)

Loops like this usually use the other major kind of loop in Java: the *while loop*. A `while` loop is much like a `for` loop, but strips out the initialize and increment parts of the loop, leaving only the conditional.

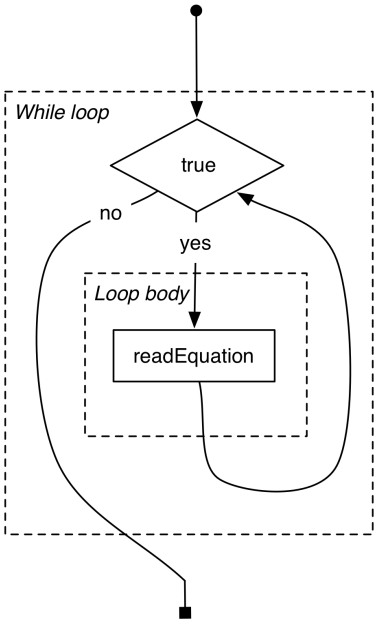
Write a simple `while` loop surrounding your new **`readEquation`** method in **`main`**. This will let your user type in one equation after another.

Figure 7.9 Write while loop

```
public static void main(String[] arguments){
    try {
        while (true) {
            readEquation();
        }
    } catch (IOException exception) {
        System.out.println("I give up! " + exception);
    } catch (RuntimeException exception) {
        System.out.println("" + exception);
    }
}
```

First things first: check out how this changes control flow. Here is a diagram of the control flow of just the `while` loop and its contents from your code above:

Figure 7.10 While loop control flow



If you compare this to the diagram of the `for` loop, you can see that you get a `while` loop if you strip off the initializer and increment steps of the `for` loop. As a result, the choice of whether to use `for` or `while` is often a matter of opinion.

Run your program to see the results:

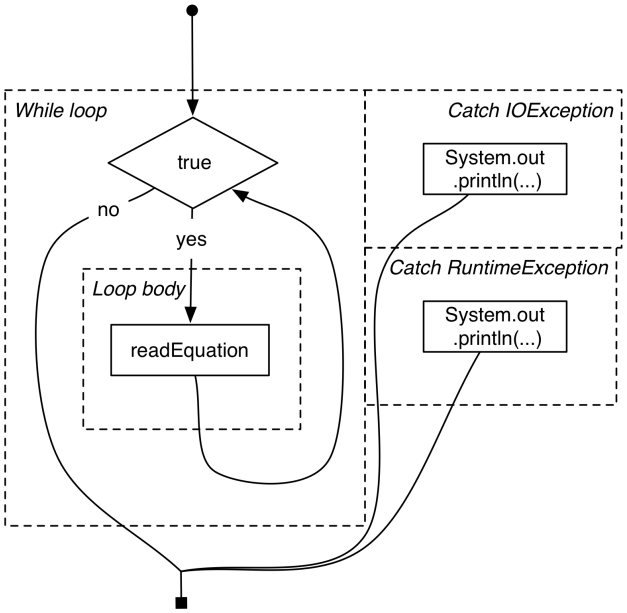
```
2
+
4
First number: 2
Operator: +
Second number: 4
Result: 6.0
6
/
2
First number: 6
Operator: /
Second number: 2
```

```
Result: 3.0
```

Your calculator app will keep on running forever, or until a user types input that your program cannot understand.

Here is your final control flow for **`main`**, including the `try-catch` blocks:

Figure 7.11 Complete control flow for main



For the More Curious: Named Breaks

The `break` statement only breaks out of the innermost loop you use it in. What if you want to break out of the outermost loop? Take this program, for example, which searches farms for golden egg laying geese.

```
for (int farmIndex = 0; farmIndex < farms; farmIndex++) {
    for (int gooseIndex = 0; gooseIndex < gooseCount(farmIndex); gooseIndex++) {
        if (gooseLaysGoldenEggs(farmIndex, gooseIndex)) {
            stealGoose(farmIndex, gooseIndex);
            break;
        }
    }
}
```

Once you steal the right goose, you want to stop searching farms, not just the specific farm you are searching right now.

Java has an obscure feature that solves this problem: the *named break*.

```
farmLoop: for (int farmIndex = 0; farmIndex < farms; farmIndex++) {
    for (int gooseIndex = 0; gooseIndex < gooseCount(farmIndex); gooseIndex++) {
        if (gooseLaysGoldenEggs(farmIndex, gooseIndex)) {
            stealGoose(farmIndex, gooseIndex);
            break farmLoop;
        }
    }
}
```

In front of the first loop, you add a label named `farmLoop`. By doing this, you can break out of the outside loop by breaking out of the labeled loop instead of the default innermost loop.

This tool is rarely necessary, but in the event you need it — now you know.

For The More Curious: Neverending Loops

The `while` loop you wrote in **`main`** has the value `true` for its conditional. That means that the only way to get out of this loop is to use a `break` statement or to throw an exception. When you want to repeat a step an indeterminate number of times, you usually use a neverending loop like this.

Neverending loops are given different names according to what they do. This loop is waiting for the user to type something in, so you might call it an *input loop*, or a *wait loop*. It is also the primary loop that the user interacts with in the program, which is often called the *run loop*.

One kind of neverending loop has a special name, though: an *infinite loop*. An infinite loop happens when you unintentionally write a loop that repeats itself before anything important happens. Like this:

```
while (true) {

}
```

If your program were to hit this loop, it would be stuck. Since it is not waiting on you for anything, it would sit there spinning and working forever, doing nothing. This is what is called an infinite loop. Programs stuck in infinite loops are commonly said to be *hung*. If you have ever tried to type text into a program and it did not respond, it might have been hung in an infinite loop.

Challenges

1. Add prompts for user input. Before reading in an equation, print out a prompt that says, “Input equation:”

2. Add the ability to gracefully exit from your calculator. If you enter the letter `q` instead of the first number of an equation, your calculator will quit.

3. Your calculator in this chapter is called a *zero look-ahead parser*. This means that when it decides what to do next, it can only look at the characters it has already read. This limitation required your calculator to do things like look for spaces at the end of numbers, because it could not read in a character and then say, “Hmm, what should I do with this?”

   For this challenge, enhance your calculator so that it does not need to read each number on its own line. After you complete the challenge, your program should be able to read input like this:

   ```
   504*76
   ```

   It will still need to look for a newline at the end of the equation.

   To do this, you will need to need to be able to look ahead to the next character and decide: “Is this a digit that I should add on to my number, or an operator that I should save?” Passing around the lookahead character will get tricky; you will probably need to alter your program significantly.

8
Arrays And Memory

So far, all of the programs you have written in this book have used local variables and parameters. It is possible to do a lot of useful work with these tools, but in practice, all Java programs use references to data in memory.

There are a variety of reasons for this, but the most important one is that the lifetime of your stack's dinner plates is just not the same as the lifetime of many kinds of data you will want to keep track of. A more sophisticated calculator could have a lot of bits of data hanging around: it might have a history of all the equations you typed in, or the ability to store an equation as a program. If that data is anywhere close to "big," there is going to be a lot of passing data back and forth between dinner plates.

The stack is a great tool, even though (perhaps *because*) it cannot do everything. Imagine if `main` were written using parameters passed on the stack:

```
    public static void main(int parameterCount, String parameter1,
            String parameter2, String parameter3, ..., String parameter64) {
        ...
    }
```

Nobody wants to write code this way, and nobody does. If a Java developer needs sixty-four of something, they never use method parameters. Instead, they use an array.

Arrays

An array is a numbered set of data. Just like variables, parameters, and return values, the data in an array has a type. Create a new command line app project called ArrayExample to play around with arrays a bit. (Remember to delete the package name in the dialog that pops up.)

In this project, much like the others you have created, IntelliJ creates a `Main.java` for you with a **`main`** method:

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

At long last, take a look at the **`String[]`** argument to your **`main`** method. The brackets on `args` mean that `args` is an array of **`String`**s, not just one of them.

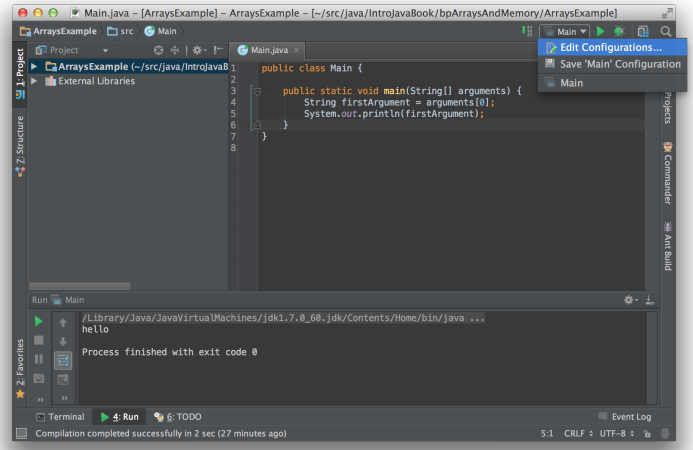
Brackets make for a good shorthand, because they are also used to access individual items in the array.

```
public static void main(String[] args) {
    String firstArgument = args[0];

    System.out.println(firstArgument);
}
```

Arguments are usually passed in when the program is run as a command line tool. From within IntelliJ it is easier to set them up inside your *run configuration*. A run configuration tells IntelliJ how to run your program. Each time you pressed the play button to test your app, you used the default run configuration that IntelliJ created for you when your project was created.

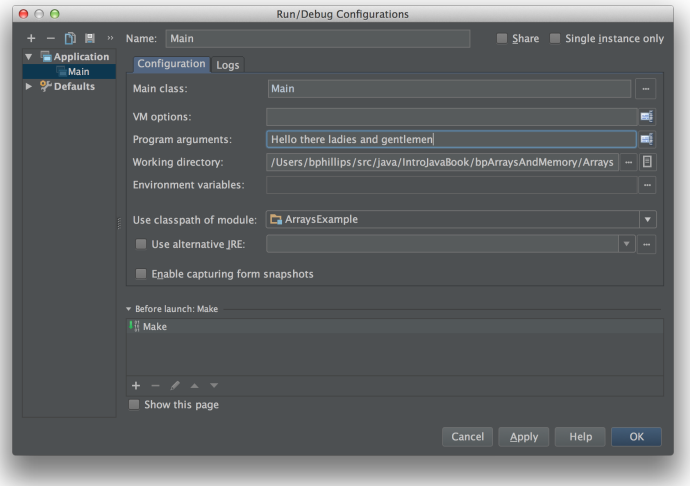
To edit your run configuration, click Main in the drop down menu at the top right, and then select Edit Configurations... from the menu that pops up.

Figure 8.1 Edit Run Configurations



Once inside there, you will see a dialog showing all your run configurations (you only have one), along with a simple editor.

Figure 8.2 Adding arguments to your run configuration



In the Program arguments: text box, type in your arguments: `Hello there ladies and gentlemen`

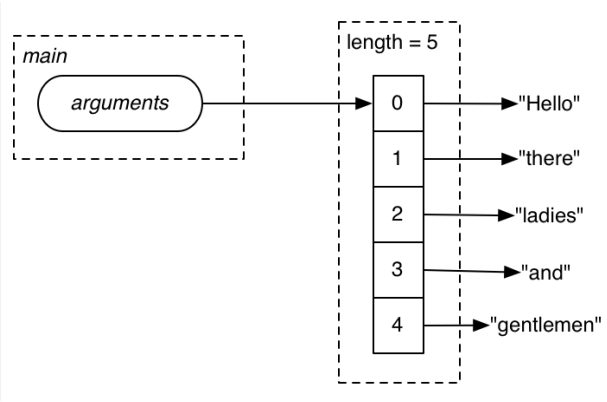
Run ArrayExample with your fixed up run configuration:

```
Hello
```

When you use brackets like this to point at something in an array, you call it “indexing” — exactly like how you point at things with your index finger. The number that you pass in is then the “index”, and the item you point to is called an *element*. So which indexes point to which elements in the array?

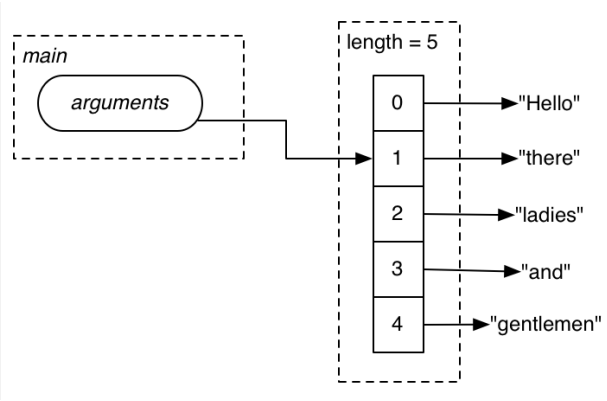
As you just saw, the first item is at the zeroth index: `args[0]`.

Figure 8.3 An Array



To point at the second item, you "offset" the pointer by one: `args[1]`.

Figure 8.4 Indexing Into An Array



If the index is too high or too low, a **`RuntimeException`** called **`ArrayIndexOutOfBoundsException`** is thrown. If you delete the arguments from your run configuration and run again, you can see this in action. Your program will produce the following output.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
    at Main.main(Main.java:4)
```

Without any arguments, the `args` array has no elements, so even zero is too high of an index. You would see the same thing with any other number you used as an index here, since the array is empty.

Fields

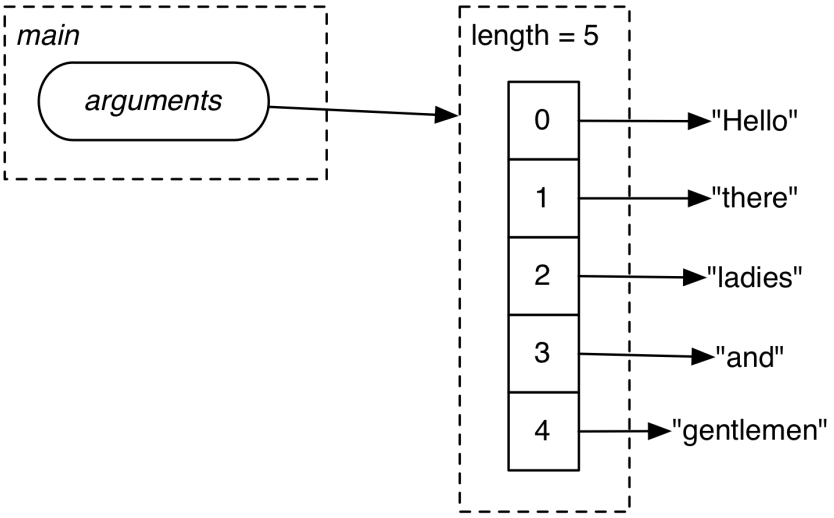
To prevent an **`ArrayIndexOutOfBoundsException`**, you need to know how many elements your array contains. Luckily, the array carries this information everywhere it goes. You can dig it out by adding `.length` to the end of your array variable. Use `args.length` to prevent **`main(String[])`** from throwing an exception.

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

```
        if (args.length >= 1) {
            String firstArgument = args[0];

            System.out.println(firstArgument);
        } else {
            System.out.println("Usage: Main <word1> <word2> ...");
        }
    }
}
```

The expression `args.length` gives you the number of elements in the array. This is called a *field lookup*, and `length` is called a *field*. A field is a named piece of data in your array. The whole array sitting in memory includes both the elements *and* and the length field. So when you point to an array, you point to the entire thing.

Figure 8.5 Your entire array



Arrays only have a single field. That is not the case for everything in Java, but it is always the case for arrays.

Iterating Over Arrays

Now that you know how to index into an array and see how long it is, you could write a for loop that iterates over the array and prints out each item.

```
for (int i = 0; i < args.length; i++) {
    System.out.println(args[i]);
}
```

If you do not need the index itself, though, you can use an easier construct called a *for-each loop*. Rewrite your **`main`** method to iterate over `args` with a `for-each` loop.

Figure 8.6 A for-each loop

```
public class Main {
    public static void main(String[] args) {
        for (String argument : args) {
            System.out.println(argument);
        }
    }
}
```

This shorthand will run once “for each” (get it?) item in the `args` array, with `argument` assigned to each value in turn.

Data In The Heap

One last thing about arrays: their data does not work the same way as locals and parameters, where all the values are copied every time you call a different method. Instead, only a *reference* is copied over. You can copy a reference a thousand times, and each reference will still point to the same piece of data.

Add some more code to `Main.java` that uses a reference. Add a new static method to change your array to say good night instead of hello:

Figure 8.7 Alter your greeting

```
public static void changeGreeting(String[] greeting) {
    if (greeting.length >= 2) {
        greeting[0] = "Goodnight";
        greeting[1] = "now";
    }
}
```

Then print out your greeting, change it, and print it out again.

Figure 8.8 Greet and degreet

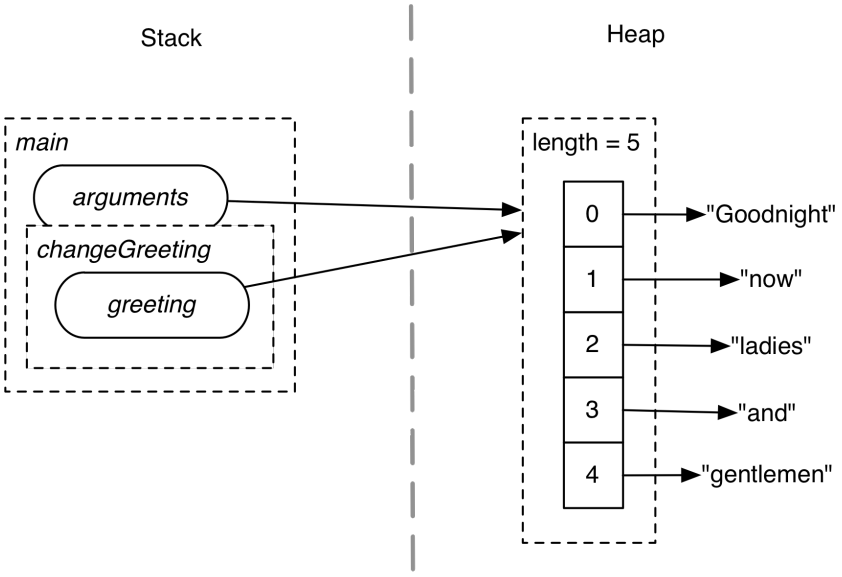
```
public static void main(String[] args) {
    for (String argument : args) {
        System.out.println(argument);
    }

    changeGreeting(args);
    System.out.println();

    for (String argument : args) {
        System.out.println(argument);
    }
}
```

Restore the old arguments in your run configuration (`Hello there ladies and gentlemen`) and run your updated program.

```
Hello
there
ladies
and
gentlemen

Goodnight
now
ladies
and
gentlemen
```

The first greeting printed out exactly what it received in `args`, like you would expect. Then you passed a reference to `args` to **`changeGreeting(String[])`**. So at the time **`changeGreeting(String[])`** was running, your program's data looked like this:

Figure 8.9 Two references



Each method had a named reference living inside its stack frame. Unlike regular values, which you can think of as boxes holding bits of data, reference values are better represented by arrows that point to other data. These references each point to a block of array data, which lives outside those stack frames.

The space outside the stack is called the *heap*. Reference values only ever point at data in the heap, never in the stack.

Other Heap Data

All of the diagrams in this chapter have showed the strings in the array living outside the array, just like how the array itself lives outside the stack frame. This is because strings live in the heap beside your array. So strings are passed around by reference, too.

Storing arrays in the heap allowed two static methods to talk to one another by pointing at the same array and modifying it. Strings, though, can never be changed. If you want a different string, you have to create a new one. Strings are stored in the heap to save time and memory that would be wasted copying large strings between stack frames.

What both strings and arrays have in common is that they are both objects. In Java, all the data that lives on the heap is stored in objects. There are many kinds of objects, and you will learn all about the different varieties of objects and how they work later on in this book.

Creating New Arrays

Let's get back to thinking about arrays. Arrays do not come from nowhere—they have to be created. You can create one in your own code, instead of taking it in from the command line:

Figure 8.10 Creating an array

```
public static void main(String[] args) {
    args = new String[5];

    for (String argument : args) {
        System.out.println(argument);
    }

    ...
}
```

The expression `new String[5]` yields a newly created array of **`String`**s that holds five elements. Arrays have the same size as long as they live, so you must specify the size of the array at the time you create it.

This is the second time you have seen the `new` keyword. You also saw it earlier in Chapter 6), because exceptions are also objects. Anytime you create an object, you use the `new` keyword to signify that you are creating new data on the heap.

Once your array is created, you can assign values to each element, just like you saw earlier. You already know what you want to go into this array, though, so you can use a shortcut:

Figure 8.11 Initializing your array

```
public static void main(String[] args) {
    args = new String[] {
        "Hello",
        "there",
        "ladies",
        "and",
        "gentlemen",
    };

    for (String argument : args) {
        System.out.println(argument);
    }

    ...
}
```

Run **`Main`** one more time, this time without any arguments. You should see the same output as you did earlier in this chapter.

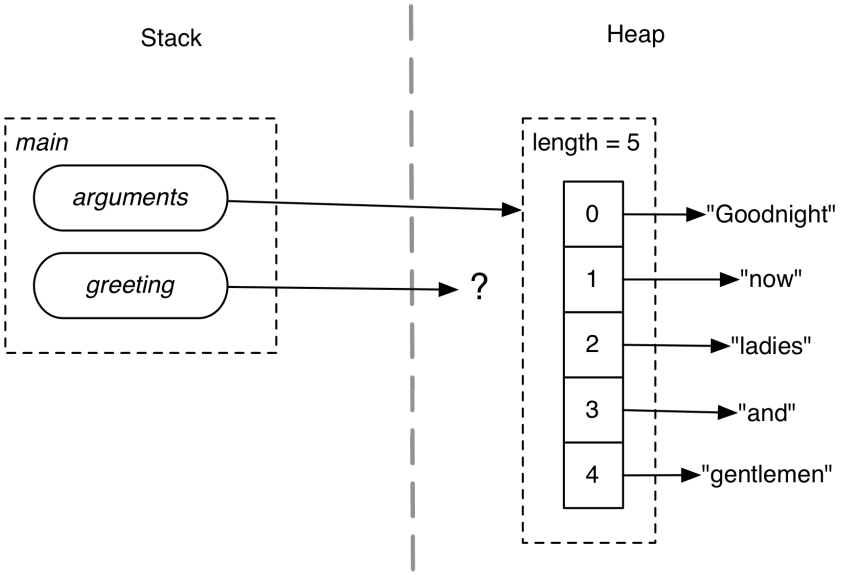
Notice that the code in Figure 8.11 does not say how big the array is. The compiler can figure that out for you from the five strings inside of the brackets, so you can leave it out.

Notice as well that you include a comma after the last item. This looks odd (you would never write a sentence that way, after all), but we prefer to write initializers with a trailing comma like this. If you decide to cut and paste that last element to an earlier position in the sequence, you do not need to fiddle around with the commas.

References To Nothing

Remember that reference variables are like arrows pointing at data in the heap. Even brand new reference variables have a space on the stack for their arrow to live in, before they have even been assigned.

Figure 8.12 Pointing at nothing



A reference pointing at nothing like this is actually pointing at a specific Java value called *null*. `null` means the same thing as nothing, except you can point to it. There is no data at `null`, and if you try to use `null` to do anything except compare it to other reference values, your program will throw an exception.

You can think of references as initially having a value of `null`. Like other primitive values, null reference values cannot be used before they are assigned. (This is not the case for data inside your array, but it does apply for an unassigned array variable.) You can always assign `null` to a reference value yourself to see what happens, though.

Figure 8.13 Using null references

```
public static void main(String[] args) {
    args = null;

    for (String argument : args) {
        System.out.println(argument);
    }

    changeGreeting(args);
    System.out.println();

    for (String argument : args) {
        System.out.println(argument);
    }
}
```

Remember that `null` references cannot be used for anything. So when you try to iterate over `args` when it is null, you will get a **`NullPointerException`**.

```
Exception in thread "main" java.lang.NullPointerException
 at Main.main(Main.java:5)
```

Good Java programmers only use null for two purposes: to say they have nothing, and to make sure that what they have is not equal to null. Any other use deserves a firmly raised eyebrow.

For The More Curious: For vs. For-Each

In most cases, there is no reason to prefer a regular `for` loop over the `for-each` loop. When iterating over objects, however, `for-each` will create and use an **`Iterator`** object.

`Iterator`s are how collection objects (covered later in this book) allow you to walk over their elements without knowing anything about the structure of the collection. This is nice, because it means that anything that implements the **`Iterable`** interface and provides a functioning **`Iterator`** will work with the `for-each` loop.

It does require one additional object to be created, though: the **`Iterator`** itself. For some performance-critical scenarios, it may be more efficient to use a plain old `for` loop, which will skip the creation of the iterator.