
Doing Stuff
With
Java
Lee S. Barney

Dedication

This book is dedicated to my wife and children. They make everything worth while. Also to those students who learn because learning is fun.

Other

Information has been obtained by Author from sources believed to be reliable. However, because of the possibility of human or mechanical error by these sources, Author, the editor, or others, Author and the editor do not guarantee the accuracy, adequacy, or completeness of any information included in this work and are not responsible for any errors or omissions or the results obtained from the use of such information.

Question and Arrow Left icons used were created by snap2objects and used under license.
Pig image used was created by Martin Berube and used under license.
Sausage image created by Daily Overview, http://www.dailyoverview.com, and used with permission.
Green check icon was created by VisualPharm and used under license.
White Zombie image used with permission of Plasmaboy Racing.
Snow goose image used with permission of Steve Perry, http://www.backcountrygallery.com.

Snow goose flock image is used with permission.

READ ME. NO, REALLY!

The book is not designed as an exhaustive API or software engineering book, a book of code script-lets to copy, nor a book full of detailed definitions of every possible word since this type of information is readily available on-line. Use the dictionary in your electronic book reader, Google®, and maybe even WikiPedia for definitions that are not explicitly given in the text for unfamiliar words.

The first chapter consists of a series if things you will need to know BEFORE you go on. Other chapters follow a consistent pattern.

Chapter Content Description

Each chapter consists of:

1. A brief description of the material to be covered and why knowledge of this information is important to you.
2. A series of points for consideration as you read and absorb the detailed information. These points are excellent items to discuss with co-workers or team members. Such discussions between peers yields greater depth of understanding and speeds learning.
3. A detailed and well organized description of the material to be learned. This material may have full color images and short videos interspersed within it. Each of these images and videos has been carefully created to clarify difficult or complex concepts.

CHAPTER 1

Stuff You Need to Know

The sections of this chapter present information that you, as a reader, need to understand in order to be successful in learning the remainder of the information presented. Read them and use them as the road to more learning.

Required Tools

Get and Install These

1. A laptop or desktop computer
2. The latest version of the Java Java Developer Kit for your computer

Java and other programming tools can come from many sources

Many tools used to create Java applications are open source and free. You can edit and compile your applications by hand using a command line terminal and a text editor or you can use an Integrated Development Environment (IDE) such as Intellij[tm]. The examples in this book use the command line terminal approach to simplify getting started. For this book we suggest the Cygwin™ terminal (http://sourceware.org/cygwin/) for Windows™ PCs. A terminal already exists as part of every OS X™ and Linux installation that will work for what you need to do.

Regardless of what environment you chose to work in, you will need to download and install the Java Development Kit™ (JDK). You can get it directly from Oracle™, Make sure you get the latest version.

To share with teammates and to allow potential employers see what you know and can do, I suggest using git and gitHub (http://www.github.com) as a content management system. If you don't already have a gitHub account go create one.

Learn git. It is a skill that employers of programmers currently want to see in potential employees. Oh... and by the way.... most of them want to see that you know how to use it from the command line. Don't use a GUI. Pro Git is a good resource book and is available for free in PDF, mobi, and ePub formats from the git website.

Example source code for this book can be retrieved from https://github.com/yenrab/doing-stuff-with-java either using git or by downloading the zip file.

The editor I used to create the examples is Sublime 2™. You can choose to use any text editor you like such as vim or textmate.

If it is impossible for you to install a compiler on you own machine, you can use an online compiler like coding ground. This is not as good an option as having a compiler on your own machine.

Think First

Steps to Success

1. Think - find out what you don't know and learn it by playing with it (sandboxing).
2. Design - use what you have just learned to layout a solution to the problem.
3. Test - create a series of expectations for behavior that will indicate if you have been successful.
4. Create - write the code!

Think before you act

Throughout history, a common approach has been used to create any type of object in the quickest and easiest way. Why would developing software be any different? It should not.

This historic approach is cyclic. This means that you take a stab at designing and then creating your item and then go back and revise one or more decisions made during design and create it again.

This historic approach is composed of four simple steps.

Think - What does the customer want? What types of knowledge are required to create the thing the customer wants? Do I already have this knowledge? If not, where can I find this knowledge? Playing with the new knowledge in a simple set of sandbox code allows you to see how each portion of the new knowledge behaves and can be used. Example: If you don't know how to use a nail gun you should probably not attempt to use it to build a house. Maybe you should play around with it and a few types of boards instead of instantly starting to build. There is no replacement for experience.

Design - Decide how the new thing your are creating should behave, look, and interact with other things. This is true of buildings, cars, toasters, baseball bats, and any other physical item you can think of, complicated or simple. This type of planning is also vital to software development. How can you create an application if you have not yet decided what it is going to do, what it will look like, and how it will interact with other pieces of software or hardware?

Test - In this step you decide what standards your product will meet when it is done. For buildings, this step would be the national and local building codes. In software this may consist of User Interaction testing, Unit testing, Component testing, System testing, Installation testing, or just figuring out, at a detailed level, what new data should come out of your application when it is given specific sets of data. You may be thinking, "How can I create a test for something that doesn't exist yet? I don't know how it should work." If this thought or one like it passes through your mind, then you have not sufficiently completed step 1, 2, or maybe both. Go back and do them first.

Create - Begin building. Notice that this step's description starts with begin. Once you have started, it is likely that you will find weaknesses in your knowledge, design, tests, or all of these. When you do, go back and revise your results for steps 1 - 3 immediately. This does not mean 'throw away and start all over again'. It means make modifications and try again.

A poor approach taken by many software programmers goes something like this:

1. A customer asks for something.
2. The programmer starts to create it.
3. It doesn't work.
4. The programmer throws it out.
5. The programmer repeats steps 2 and 3 until it works.
6. The programmer produces something the customer didn't want.

I call this the 'oh crap!' or 'think last' approach since nearly every time the developer hits step three 'Oh crap!' or similar words are heard. The 'think last' approach all but guarantees that the software ships late, is over budget, and doesn't have the features the customer wants. It is a proven failure method. Don't use it! Use the method that is known to work. **Think, Design, Test, Create**. Until you use it it may seem like this method will take longer than the 'Oh crap!' method but it does not. Guaranteed.

Helpful Resources

Steps to Success

1. Install the tools found in the Required Tools section of this chapter.
2. Learn to launch your terminal (Cygwin terminal for Windows™ users).
3. Learn to use vim or some other text editor.

Books such as this one are not your only resource. This section has links to resources that you will find helpful as you begin programming in Java.

A video for Windows users on installing Cygwin.

A quick video of some basic terminal commands

Another quick video of some basic terminal commands

Turning on syntax highlighting and line numbering in the vim text editor

A using vim video

A git installation tutorial.

Movie 1.1 Downloading the example files

The code samples for this book can be downloaded.

Another vim tutorial video

The Java API.

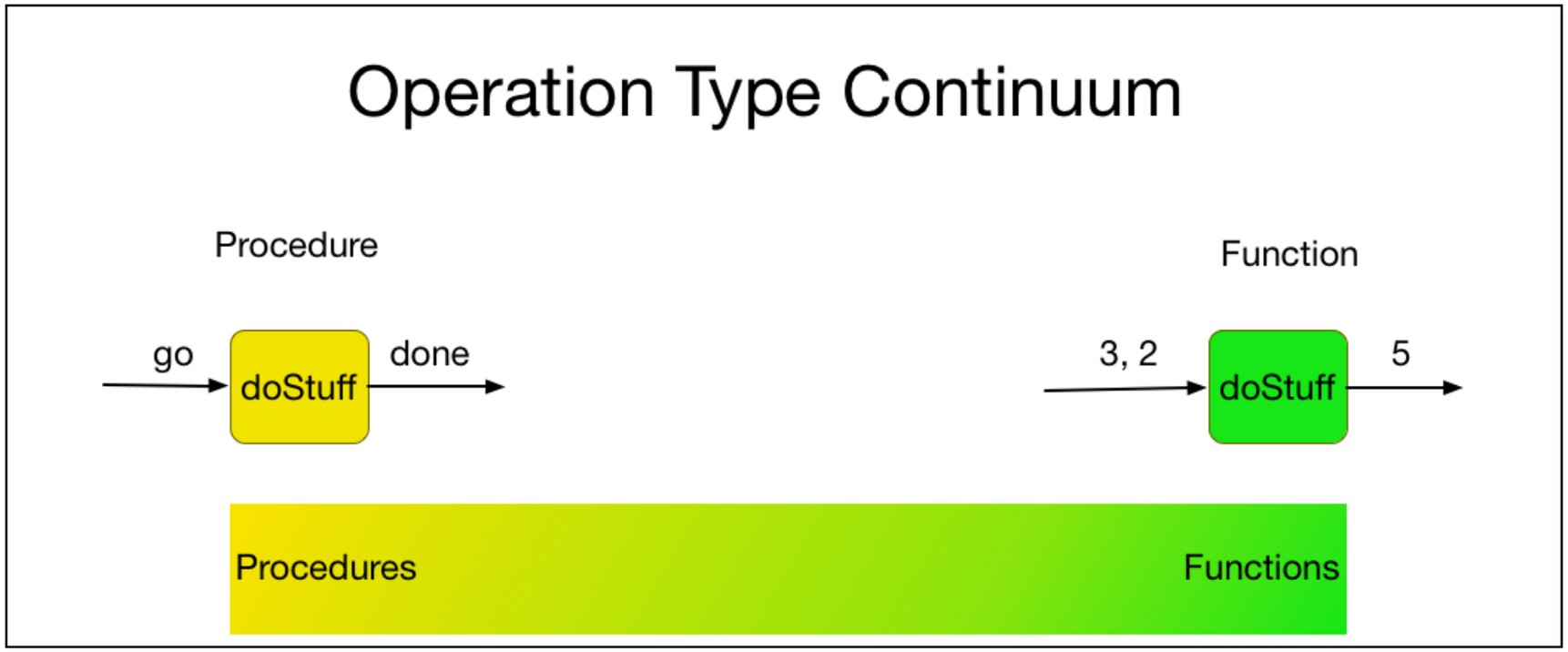
The types of variables that can be used in Java. Don't worry about the reference/Object data types. We'll cover them later.

The operators available in Java.

Operations and Languages

Points to Ponder

1. What is an operation?
2. What types of operations are there?
3. How can common computer languages be categorized?



Operations can be a pure procedure, a pure function, or somewhere in between.

I've heard some people say they like procedural programming and don't like Object-Oriented programming. I'm not sure what they mean. It seems they're saying they like 'step-by-step' programming.

All programming, regardless of the language or language type, is created and executed 'step-by-step.' All programs are a series of operations--things to be done. Each operation is executed one after the other regardless of the type of the language.

There are two basic types of operations, procedures and functions. Procedures are things to be done that require no data to begin with, do

stuff, and let you know when they are done but don't give you back any computed results. A function requires data to do its job, does stuff, and then gives you a result back when it's done.

A physical example of a procedure a request to a teenager to clean their room. There is no additional data needed. They clean their room. When they are done is obvious because they come out of their room. At least that is how it would be in an ideal world.

An example of a function would be an oven baking something. If you put in a bread dough mixture and a pan you get bread back out when the bake function is done. If you put cake dough and a pan in you get cake back. Either way, you get back some type of baked good.

With procedures and functions available, it is common to classify programming languages as procedural, using only procedures, or functional, using only functions, even though no programming language is purely procedural or functional. Some are more procedural and others more functional.

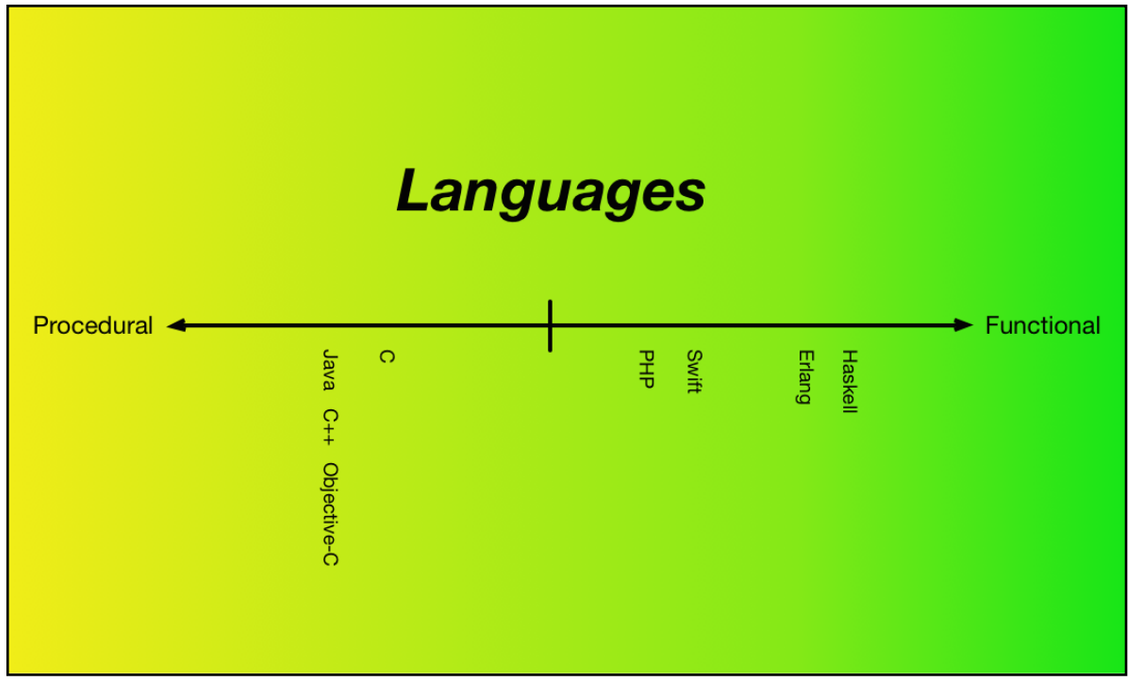


Figure 1. *A classification of some common programming languages.*

Based on the frequency of functions and procedures in the Java API, Java tends to be more procedural than functional. Using the same frequency scheme, C++ and Objective-C are about as procedural as Java. Languages such as Haskell, Erlang, and even PHP tend to be more functional than procedural.

Regardless of which type a language is, and this can vary depending on who you ask, all languages use of both procedures and functions.

CHAPTER 2

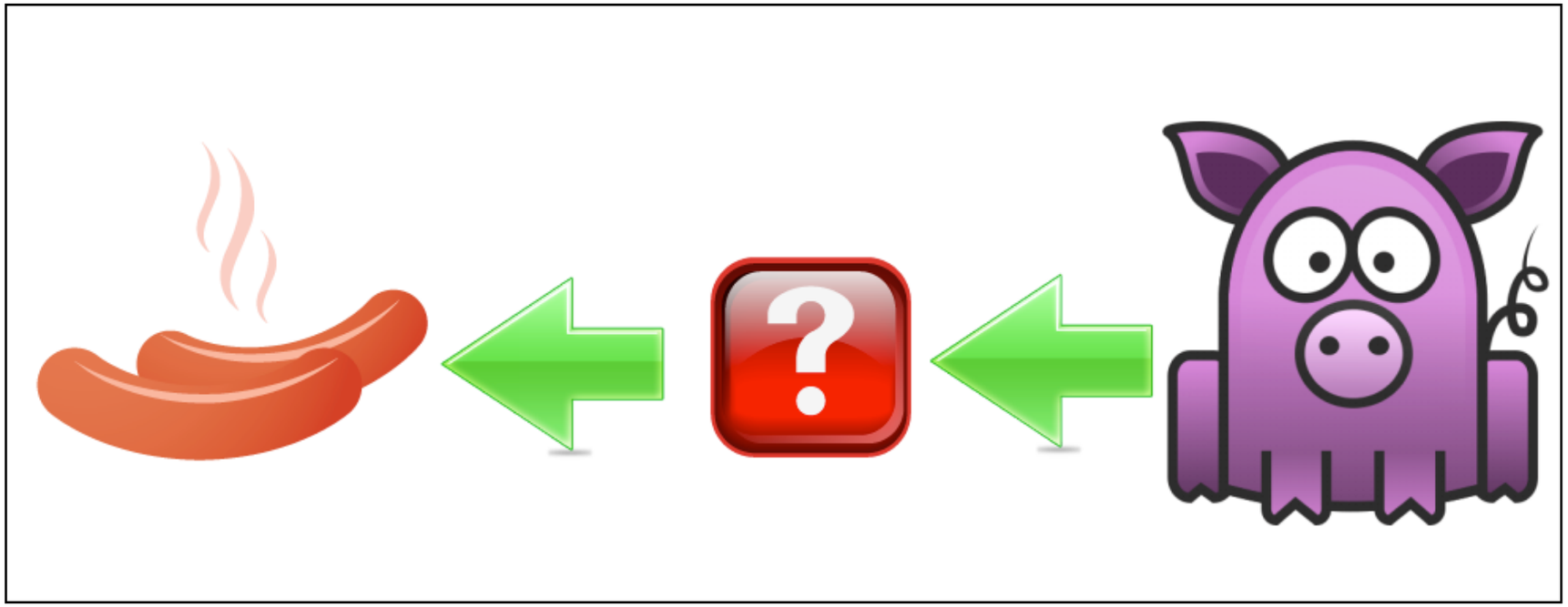
Ready...Set...Go!

White Zombie is the worlds fastest, electric, street legal, drag racing car. It REALLY goes. Some of it is custom assembled but not all. The body, frame, tires, wheels, and other components were created by people other than Plasmaboy racing. Applications you write are built the same way. Some parts are pre-made for you and others you will assemble yourself from pre-made parts.

Functions and Procedures

Points to Ponder

1. What is a function and what is its purpose?
2. What do functions look like in the Java language?
3. What is a procedure and what is its purpose?
4. What do procedures look like in the Java language?



Functions are one of the basic building block of all applications written using the Java language. Stuff goes into a function and stuff comes out. A sausage factory is a physical manifestation of a makeSausages function.

When you say you are writing an application in Java what you are really saying is that you are creating a bunch of operations and putting them in a specific order. You are telling the person you are speaking with that you are going to arrange these operations so you can effectively communicate with the user of your application and help them accomplish something.

Do NOT fall into the common error of thinking your application will talk to the computer. The computer is the medium of your communication with the user -- just as air is the medium when you use English or some other language to speak with someone.

Since procedures and functions are types of operations and are the foundations of everything

you will do when you use the Java language we begin there. Answers to the questions for this section will be provided and the details of procedures and functions will be examined. Hold on to your hat. At times you may feel like a pig in a sausage factory but you won't get ground up and spit out. I promise.

The grammar of the Java language, usually called 'the syntax' by techies, declares that functions and procedures all have a similar structure, they have a name, a group they belong to, optional stuff that goes in, optional stuff that comes out, and stuff that makes up what the function or procedure does.

For a pork sausage factory function the structure would consist of

1. a name - makePorkSausages,
2. a group - FactoryWork,
3. stuff that goes in - pigs,
4. stuff that comes out - pork sausages, and
5. stuff that makes up what the factory does - chopping, filling, and a bunch of stuff no one really wants to know about.

Take the time right now to start thinking of your everyday tasks. Each of them could be described as a function or procedure. An example could be taking a shower or bath. You, soap, shampoo, water, a tub or shower, and the grime you have built up on you over time goes in. What comes out? What do you do that makes up the process of taking a bath or shower?

If you start to examine what you do during any given day you will find functions and procedures everywhere. I strongly advise you start doing this NOW. In every spare moment you have look around you. See what people, animals, buildings, cars, tools, bugs, plants, clouds, etc. are doing. Truly look and think.

Give what you see going on a name. What group of functions or procedures could it belong to? Figure out what, if anything, is going in, what, if anything, is coming out, and what may be happening as part of what is being done. Feel free to write down what you observe. It may help you later if you start becoming confused as we talk specifics. If you start examining your world now in every spare moment in this way, learning the Java language will be MUCH easier.

Before reading this book any further go do this for at least two days. ***STOP HERE. PLEASE.*** Don't make your life harder than it has to be.

Where To Begin?

Now that you've spent some days looking for functions, procedures, and their parts, let's start up again by taking a look at a very special procedure that is part of the Java language. Its name is 'main'. In the Java language, the main procedure exists for one purpose only. When you tell a computer to start an application it has to know where to begin executing your program. In the Java language this is the purpose of the main function. The program that runs your Java code will look for this specific procedure, find it, and execute it. If it isn't there the computer won't know where to begin.

Let's see what this special procedure looks like.

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

Like I said earlier, the Java language's grammar, its syntax, states that a function must have 'stuff that comes out'. The `main` procedure, according to what you see above, declares it expects nothing to come out. That is what `void` means. This is why `main` is a procedure. I'll show you how to make stuff come out of functions later.

Don't worry about the 'public static' part. It's used to state who can use it and how to use it. We'll cover what these mean and do after you've gotten used to Java.

The other strange things that you see are the parentheses, `()`, after the `main` function's name. These parentheses are used to hold the 'stuff that goes in'. This example of main has has a bunch optional of strings going in so there is that 'String[] args' stuff between the parenthesis.

All functions and procedures in Java follow a basic grammar rule. All Java functions look like this.

whoCanUseIt howToUseIt whatComesOut theFunctionName(whatGoesIn)

All java procedures look like this.

whoCanUseIt howToUseIt nothingComesOut theFunctionName(whatGoesIn)

For procedures, the 'whatGoesIn' is optional.

You can depend on this rule always being true. What you haven't seen yet is 'the stuff that makes the function or procedure do what it does'. We'll add that in now.

```
public static void main(String[] args){
        System.out.printf("%d\n",3 + 5)
}
```

Details:

In the Java language's grammar, the characters `{` and `}` are used to contain 'the stuff that makes the function or procedure do what it does.' They are used to help us humans see, understand, and remember what this function does compared to other functions that may be hanging around in the same location.

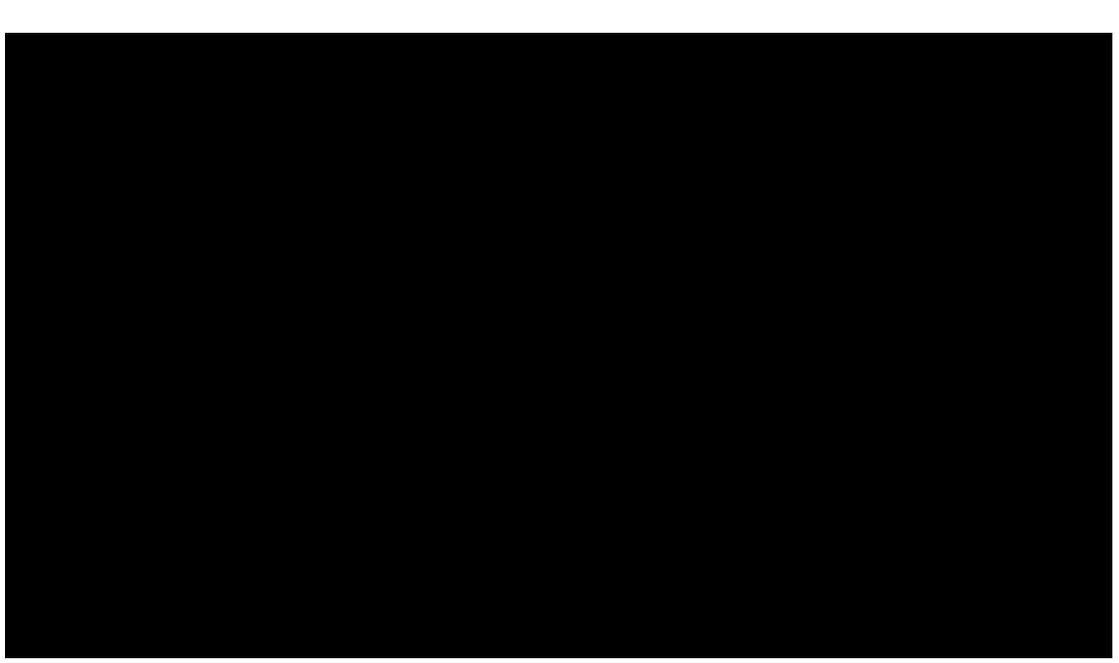
Great debates still swirl around the internet about exactly where is the 'best' place to put the `{` and `}` characters in spite of the debate being over 40 years old. Forty years is WAY too long for debating such an unimportant thing. On the internet and in books you will see examples with these characters in different places based ONLY on the author's opinion. All of the examples you will see from me will follow the pattern you see above.

Now that that's out of the way, lets examine what the parts that make up what this version of the main procedure is going to do when it runs. We are going to use another special procedure, printf. The printf function belongs to the System.out group of

functions and procedures. It tells your computer to print a line to the terminal. When your application runs, the number 8 will appear. Don't worry too much about understanding printf right now, we'll discuss it in detail a little later.

There you have it. You now have seen all the basic parts of functions and procedures. All of the other functions and procedures you'll ever write using the Java language will have these same parts and follow the same grammar. Go download this main method example by getting it from the gitHub repository of examples.

Movie 2.1 Compiling and Running the ***first_hello*** application



Compiling what you have written in the Java language is required in order for your computer create and then run your application.

Now I'll give you a freebie. This video shows you how to compile this source code into an application, run it, and see the result.

Enough of using other people's special procedures! Let's create a function of our own. Let's have it do something that you have been doing since you were small--multiply two numbers together.

Remember when I told you I'd show you how to make stuff come out later? It's later. `return` is a special word in the Java language. It means 'take what is coming next and make it be what comes out of my function'. In the FirstFunction.java code sample, a product is what will come out of your *multiply* function.

Let's start by putting your *multiply* function in a new group and call this group 'FirstFunction.' There is no special reason for this group name. You can name your groups anything you want. A couple of restrictions are:

• the group name can't have spaces in it, and

•The group name must match the name of the file your code is in with '.java' added at to the end of the file name.

We'll put this code in a file called FirstFunction.java. It's in the gitHub repository of examples.

```
/*
 * Here is the brand new function multiply. It is in the
FirstFunction group.
 */

1  public class FirstFunction{

2     public static int multiply(int firstNum, int secondNum){
3         return firstNum * secondNum;
4     }
5
6     public static void main(String[] args){
7         int product = FirstFunction.multiply(3, 4);
8         System.out.printf("3 * 4 = %d\n",product);
9     }
10 }
```

Code Sample - FirstFunction.java

Let's start by describing our new *multiply* function. It is on line number **2** and has all the parts of functions we talked about before--who can use it, how to use it, what comes out (an int), the function name (multiply), and what goes in (two ints, firstNum and secondNum). Again, don't get all worried about 'public static' right now.

What's an int?

Details:

Line number **3** says "After you multiply what ever firstNum and secondNum happen to be, send the result back out of the function." Using the *multiply* function we just created is much like writing the FirstHello example. Line **7** shows the *multiply* function being used. Line **7** reads, "From the FirstFunction group use the multiply function to multiply three and four and keep what is sent back as an int called *product*.

Take a breath. That was a lot of stuff. Spend some time going over the description of line **7** and comparing it with the description. It usually takes a bunch of attempts to get to where you understand the description.

Now let's go on. Line **8** uses the standard Java printf function found in the System.out group. It is ugly. What it ends up in the terminal is "3 * 4 = 12" but "what the heck does all that %d\n stuff do?" In order for printf to print out the integer held in product we have to tell it what what kind of thing product is. That is the purpose of *%d*. The d stands for decimal. Yep, your right. Integers don't have decimal points in them so why isn't it %i for integer? Queue some old guys with the beards. Long ago they chose decimal for its other meaning. I can almost hear them saying, "Decimal means base 1. Let's use %d for regular old numbers. That way we could also do cool stuff like hexidecimal numbers!"

Oohh.. Ahh... not really all that cool.

The *\n* part stands for "make the next thing printed out in the terminal show up on its own line. If you leave this out everything will come out on the same line and your terminal will start looking really funky. Make sure you don't forget it.

So there is our first created function. It doesn't do anything important but I hope you get the idea. Don't forget what functions and procedures are or how they work. We need them all the time when working in Java.

"But wait a minute!" you say. If printf is a function it is supposed to give us something back! Be calm. It does but you don't want to worry about that right now. We'll just ignore what we get back and it will go away.

Remembering User's Info

Points to Ponder

1. How can I store data so I don't loose it when I stop the application?
2. How do I get data back into my application once it has been stored?
3. What can I do to allow the user to communicate?

Information is vital to our lives. Even small choices like where to go on a hike require information. Be ready for a workout if you hike Damnation Creek Trail.

When you write an application you are communicating with the user. For there to be true communication it must be bidirectional. The user needs some way to speak to you.

Another point to remember is computers are stupid. They can sometimes seem smart because they are so fast at being stupid. They can make all kinds of stupid mistakes, evaluate and reject them, and then finally make a good choice.

Because of their stupidity, computers can't remember anything. When you stop an

application anything it has been working on is gone...unless the application is designed to store what it has done.

In computer terminology the information stored or shared with the user is called output. Anything coming in from where it was stored or from the user is called input. Together these are called IO, Input-Output.

One way to generate output is using the System group's printf function you saw in the last chapter (**HINT: If you haven't read chapters 1 and 2, GO BACK!!!**). Being the highly intelligent person that you are, you have assumed if there is a way to generate output there must be a way to get input; and you would be right. When you want to get knowledge out of your head and into a book for later use by yourself and others you get the information printed. When you want to get information back out of a book and into your brain you read it.

So being intelligent you would assume that the input partner to printf would be readf. But the white haired, long bearded guys that created Java's IO functions weren't as bright as you. I can almost hear them. "Nobody reads books anymore and they certainly won't in the future! At best we just scan 'em and look for interesting things."

So they decided to call System.out's printf's partner for reading ints 'nextInt' and put it in a group called Scanner. I know...I know. Confusing decision but we are stuck with it.

Not thinking that choosing a bad name was enough, they decided to make Scanner confusing to use. Did they create Scanner.in.nextIntf? Nope. How about System.in.nextInt? Nope. They did manage to make Scanner read in only one thing at a time, thank heavens.

Here is an example that reads in two ints. They are then passed to the multiply function you created in the last chapter (See...I warned you to go back and read that one.) To make things simpler, we'll create a new group called FirstFromUser.

```
5   public static void main(String[] args) throws Exception{
6     Scanner systemInScanner = new Scanner(System.in);
7     System.out.printf("enter an integer: ");
8     int firstInt = systemInScanner.nextInt();
9     System.out.printf("enter another integer");
10    int secondInt = systemInScanner.nextInt();
11    int product = FirstFunction.multiply(firstInt, secondInt);
12    System.out.printf("%d * %d =
                                %d\n",firstInt,secondInt,product);
13 }
```

Code Sample - FirstFromUser.java

Details:

OK, OK... don't panic. That's the most important thing right now. DON'T PANIC. I'll step you through the FirstFromUser.java code sample.

In order to use the code from the Scanner group we are forced to first make a brand new Scanner. That's what's happening on line 2. You'll see the word new there. 'new Scanner(System.in)' means "make me a new Scanner that can read from the System.in group." The System.in group represents your terminal. Lines 3 and 5 look familiar from the last chapter. They tell the user to enter an integer of their choice.

What is 'throws Exception'?

Lines 4 and 6 are the nasty ones. The nextInt() function of the Scanner created on line 2 is used to get and store the integer the user types in. Since we're reading something in, Java makes us keep it somewhere. We put them in *firstInt*, line 4, and *secondInt*, line 6.

Line 9 uses multiple %d's to print out the two integers the user typed in, held by *firstInt* and *secondInt*, and the calculated result held by *product*.

This is all fine and dandy, but what about storing the users info? That's what this chapter's about right?

Storing Data:

In Java information is stored using what are called files. 40 years ago keeping paper files in file cabinets was very common. The writers of C, one of Java's ancestors, decided to use an analogy to those files and file cabinets that everybody used back then. They did this to help them relate to what the computer was doing. The analogy made sense 40 years ago but not so much today. Most of us have never had a paper file that we put it in a filing cabinet.

Today we might choose a different analogy for information storage. Documents are what we tend to work with today. If you can remember that a file in Java means a document we are good to go.

In Java, each file is somewhere on the computer. In Java, a file's location is represented by its path. This path ends with the name of the file (Ex. *C:\allMyStuff\user.data*). We can also represent a file just by the name of the file (Ex. *user.data*). When we do, the file will end up in the current directory--the directory we have moved to while using our computer's terminal.

Let's write our very own procedure, part of the WriteToFile.java code sample, and make it save the information we got from the user. To do this we'll need to use two new groups, *PrintWriter* and *File*.

```
15 public static void saveNumbers
                (int aNumber, int anotherNumber) throws Exception{
16    File dataFile = new File("user.data");
17    PrintWriter fileWriter = new PrintWriter(dataFile);
18    fileWriter.printf("%d %d",aNumber, anotherNumber);
19    fileWriter.flush();
20 }
```

Code Snippet 1

Details:

Line 15 in Code Snippet 1 says, "Create a procedure named *saveNumbers*. It should expect to be passed two ints, aNumber and

anotherNumber." *saveNumbers* is a procedure since it returns nothing. Remember this from Section 1?

Line **16** uses one of our new groups, File. This group is used by Java to represent the location in your computer of your *user.data* file. We'll need to use File later so you'll want to remember it. Like Scanner, we have to create a new version of the File group in order to use it. That is why you see 'new File' and then the file name, *user.data* as part of line **16**. If this is confusing, please go back and review when we created a version of Scanner in the FirstFromUser.java code sample.

Line **17** of Code Snippet 1 reads, "Create a version of the PrintWriter group that can write to the file remembered by the 'dataFile' File." PrintWriter versions work like the System.out group. That's why we are going to use them here. It reduces the amount of new stuff we have to learn, thank heavens. You'll see why as we look at the next bit of code.

Line **18** is similar to when we used printf to tell the user the product in the FirstFromUser example . This time the line says, "Use the fileWriter group's printf function to write out an int, a space, and another int. The values to be written out are the ints stored in aNumber and anotherNumber."

Be careful! Using printf won't add to an existing file. It will delete the file and start it over from scratch. If you want to see how to add to what is in a file take a look here. Beware, it's more complicated.

One more line to go and were through the worst of it. Since we wrote to a file we should be good people and flush what we've written. If we don't then, strangely enough, the writing won't happen. Remember, anything you write you must flush to get it to be in the file.

Now that we've got a *saveNumbers* procedure, let's use it to store some numbers. To keep things simple we'll reuse the code from the previous example where we read in two integers from the user.

```
6  public static void main(String[] args) throws Exception{
7     Scanner systemInScanner = new Scanner(System.in);
8     System.out.printf("enter an integer: ");
9     int firstInt = systemInScanner.nextInt();
10    System.out.printf("enter another integer");
11    int secondInt = systemInScanner.nextInt();
12    WriteToFile.saveNumbers(firstInt,secondInt);
13 }
```

Code Snippet 2

Line **12** contains the call to *saveNumbers* instead of the call to *printf* and multiply that were there before. That's it! We've saved information from the user!

Retrieving Data:

OK...now how do we get these ints back? That's the point of putting them in the file, right? So the computer can 'remember' them.

To get them back we'll create another function, part of the ReadFromFile.java code sample. We called our saving procedure *saveNumbers*. Let's call this one *readNumbers*. That way the name will reflect what it does and mirror the way we named *saveNumber*. *readNumber* is part of the *ReadFromFile.java* example and has all of the parts of a function, the who can and how to use it indicators, the stuff that comes out, its function name, the stuff that goes in, and the what it will do. If you've struggling with these function parts, PLEASE go back to Section 1 and review what functions and parameters, the stuff that goes in, are.

```
17  public static int readNumber(Scanner fileScanner)
                                                  throws Exception{
18      int aNumber = fileScanner.nextInt();
19      return aNumber;
20   }
```

Code Snippet 3

The *readNumbers* function has one thing going in, a version of the Scanner group named *fileScanner*. Why *fileScanner*? Because I think that name describes what it is going to do. We will use it to scan the user.data file for the ints we stored there in the WriteToFile.java code sample. The truth is, you can name things anything you want but for simplicity sake I always try to name things what they are. It makes it easier for me, and others, to read my code.

Line **17** says, "Create a function called readNumber that expects to get a Scanner named *fileScanner* and will put out an int." In line **18** the fileScanner group's nextInt function is used to store the next integer from the *user.data* file in the *aNumber* int. Then line **19** uses the return word, which we've seen earlier, to send the value stored in aNumber back out.
OK...fine....you say, "But why aren't I seeing the name of the file, *user.data*, anywhere? It must have to show up somewhere or the computer would have to be smart. But computers are stupid, right?"

Yep. Your absolutely right. Somehow we, the smart ones, have to tell the computer to use the *user.data* file just like we did when we put the ints in the file.

Hopefully you remember the File group we used in the WriteToFile.java code sample because, you guessed it, we're going to use it again!

```
7  public static void main(String[] args) throws Exception{
8     File userDataFile = new File("user.data");
9     Scanner fileScanner = new Scanner(userDataFile);
10    int firstNumber = ReadFromFile.readNumber(fileScanner);
11    int secondNumber = ReadFromFile.readNumber(fileScanner);
12    System.out.printf("%d * %d = %d\n",
                                  firstNumber,secondNumber,
                                  firstNumber * secondNumber);
13    System.out.printf("%d + %d = %d\n",
                                  firstNumber,secondNumber,
                                  firstNumber + secondNumber);
14    System.out.printf("%d - %d = %d\n",
                                  firstNumber,secondNumber,
                                  firstNumber - secondNumber);
15 }
```

Code Snippet 4

In Code Snippet 4 we are in the *main* method again. Hopefully you're becoming more familiar with the main method. Line **8** is where a version of the File group is created and "user.data" is used. This is the same as when we used it to write to the file. Line **9** is where the... fun begins. Remember back in Section 1 when you used the Scanner group to get information from the user? We get to use it again here. A version of the Scanner group is created on line **9**. Line **9** reads, "Create a version of the Scanner group called fileScanner that can scan the userDataFile." Now all we need to do is send fileScanner to ReadFromFile's readNumber function and we'll be done! Since we want two numbers out of the file we use readNumber twice, lines **10** and **11**. Lines **12** , **13**, and **14** are where the code tells the user what was found and what the product, sum, and difference are.

That's it! We are done! With these examples your program should be able to remember, in other words read and write, information gathered from the user or calculated as part of an application. Use this in your apps. Nobody wants to use an app that can't remember what they already told it.

Making Choices

Points to Ponder

1. How do C applications choose between different behaviors depending on the situation?
2. How do I structure the choices so my application's behavior is predictable?

If you go out on a limb you will eventually have to choose a branch to follow.

Choosing between things is always difficult. Choosing one thing means not choosing the other others. Studies have shown that the more options humans have the worse they feel about the choice they made. Think of the last time you bought ketchup. So many options.

Computers don't have the problem of not being satisfied with the choice they make. But then again, they're stupid.

Java has a structure that allows code to choose between few or many options. Being based on English, Java uses english words to express choice making. In English we could phrase a

choice between multiple things by saying 'If ....otherwise if....otherwise if....otherwise.' In Java it is written 'if....else if....else if....else.'

If what? How is the 'what' portion described in Java? If we wanted to we could check to see if a number is even or odd. To do this we need to use the Modulo operator, %. It returns the remainder when two integers are divided. 5%2 yields 1, 4%2 yields 0, 3%2 yields 1, and 44%2 yeilds 0. So any number %2 is even if the result is 0. Otherwise the number is odd.

```
1  public class EvenChoices {
2
3
4    public static void main(String[] args) throws Exception{
5        checkInts();
7     }
8     public static void checkInts() throws Exception{
9        int aNumber = 5;
10       int remainder = 5 % 2;
11       if(aNumber == 0){
12            System.out.printf("0 is neither even nor odd.\n");
13       }
14       else if(remainder == 0){
15            System.out.printf("%d is even.\n", aNumber);
16       }
17       else{
18            System.out.printf("%d is odd.\n", aNumber);
19       }
20   }
18 }
```

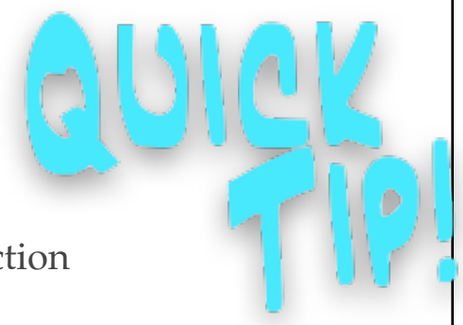
Code Sample - EvenChoices.java

Details:

The EvenChoices.java code sample checks a single number, 5, for being odd or even. Lines **11**, **14**, and **17** are the specific lines where this check occurs. Line **10** calculates the remainder. Line **11** is a special check to see if the full number, *aNumber,* is zero. It reads, "If aNumber is equal to zero." Just like with function declarations, what happens in this case is contained within the { } characters.

Line **11** reads, "else if the remainder is equal to zero." This is the second alternative. The third alternative is declared with the *else* keyword. Look CAREFULLY. The else keyword doesn't have any check associated with it! It will be where everything goes that doesn't fall into the groups described by the checks on lines **11** and **14**. In other words, if aNumber isn't zero and isn't even, then line **17** will kick in.

But what if I don't want to do a check for something being equal? That can be done the same way using one of Java's comparison relational operators.



The FractionChoices.java code sample checks to see if any positive double is a fraction between -1 and 0 and between 0 and 1.

What's a double?

```
1  public class FractionChoices {
2
3
4    public static void main(String[] args){
5        checkDoubles();
6    }
7
8    public static void checkDoubles() throws Exception{
9        double aNumber = 0.25;
10       if (aNumber > 0 && aNumber < 1){
11        System.out.printf("%f is only fractional.\n", aNumber);
12       }
13       else if (aNumber < 0 && aNumber > -1){
14        System.out.printf("%f is only fractional.\n", aNumber);
15       }
16       else{
17        System.out.printf("%f has a whole number component.\n"
                          , aNumber);
18       }
19   }
20 }
```

Code Sample - FractionChoices.java

Java doesn't have a *between* operator so you need to do two comparisons to check if something is between two numbers. Line **10** reads, "If aNumber is greater than zero and aNumber is less than 1." The && characters one of the logical operators available to you in Java. It stands for 'and.' It is very commonly used so you should get to know it well. Line **13** does an additional check. This one is looking for purely fractional negative numbers. It reads, "else if aNumber is less than zero and aNumber is greater than negative 1." If aNumber doesn't match line **10** or line **13**, then line **16**, the default choice, will execute because there is a whole number part to *aNumber*. That is, it must be zero, less than or equal to -1, or greater than or equal to 1.

The FractionChoices.java code sample has a problem. Lines **11** and **14** are identical. In the world of programming this is called code duplication.

Code duplication causes problems. Not only is it inefficient to write, but it can wreak havoc when you start to fix bugs. Take my advice. Don't duplicate code.

"But how," you ask, "can FractionChoices.java be done without code duplication?" Excellent question. Put both checks in one *if* statement.

```
1  public class FractionChoices2 {
2
3
4    public static void main(String[] args){
5        checkDoubles();
6    }
7
8    public static void checkDoubles() throws Exception{
9        double aNumber = 0.25;
10       if ((aNumber > 0 && aNumber < 1)
                    || (aNumber < 0 && aNumber > -1)){
14        System.out.printf("%f is only fractional.\n", aNumber);
15       }
16       else{
17        System.out. printf("%f has a whole number component.\n"
                          , aNumber);
18       }
19   }
20 }
```

Code Sample - FractionChoices2.java

Details:

Line **10** of FractionChoices2.java is different. It now reads, “if aNumber is greater than zero and aNumber is less than one **or** aNumber is less than zero and greater than -1.” This is a compound check. The || characters represent the idea of **or**. The two aNumber checks have been put in their own sets of parenthesis. This keeps our poor, stupid computer from getting confused. Without them it can get all mixed up. Better safe than sorry. No one likes to debug vague code. Write your code exactly how you mean it and when you come back to fix it later any thinking error you made should be easier to find. If you write your code vaguely, you won’t remember what you meant to do and will waste a lot of time. The extra parenthesis help your code not be vague.

Groups and Loops

Lorem Ipsum

1. What is an array?
2. How do you make an array?
3. How do you use an array?
4. What are two of Java's types of loops and when are they used?

In Java all elements of a group of things, called an array, must be of the same type just like this flock of geese.

So what if we need to represent a bunch of numbers? Maybe we are averaging two years of monthly income from our awesome application. Is there a way to do that without creating a whole bunch of doubles? Yep. There is. You'd think someone thought through this before.

In Java, groups of things, like the income from your app by month, are called arrays. This is the only native grouping in Java though you could write code to create other types like lists, dictionaries, and trees if you wanted to. We'll stick with arrays in this book.

Arrays are both great and dangerous. Misuse of arrays in Java can cause your code to be hard to support and even cause it to quit before you expect it to. We'll talk about that later, but remember...you MUST be very careful when using arrays.

OK. So lets get started.

```
9  double numbers = {0.25, 11.003, -2.14, 0.14,
                                   20001.3, -.000001, -10.0};
```

Code Snippet 1

Line 9 in Codet Snippet 1, part of the GroupLoop.java code sample, shows how to make an array. It reads a little backwards so you might need to compare this to the code multiple times. "Create an array of seven doubles called numbers and put 0.25, 11.003, -2.14, 0.14, 20001.3, -.00001, and -10.0 in the array." Wow. The left hand side of the = sign sure is mixed up. Why doesn't it look more like this?

```
   array 7 double numbers = {0.25, 11.003, -2.14, 0.14,
                                   20001.3, -.000001, -10.0};
```

Bad Code

Java is designed to use few characters in situations like this one. The guys that designed Java preferred what you see on line 9 of the GroupLoop.java code sample. When you write your own language you can decide how it will look. Until then you are stuck with other people's decisions.

So line 9 shows how to put an array together. Any time we put something in a group we eventually need to get something out of the group. When you put groups of dollars, euros, or what ever your currency is into a bank don't you eventually intend to take them back out?

How then can we get a value out of an array? Code Snippet 2 shows how.

```
double aNumber = numbers[1];
```

Code Snippet 2

It reads, "the double aNumber is the same as the one'th value in the numbers array." Now you may think aNumber would now be 0.25. That is a very logical conclusion. It is actually 11.003. What?!

Notice I wrote "one'th" rather than "first." Hmm... tricky. Arrays in Java represent their first position with a zero rather than a one. Code Snippet 3 shows aNumber becoming 0.25.

```
double aNumber = numbers[0];
```

Code Snippet 3

This means that even though there are seven elements in the numbers array, you assign the last element of the numbers array, -10.0, to aNumber is done using a 6. Code Snippet 4 shows you how to do this.

```
double aNumber = numbers[6];
```

Code Snippet 4

Weird...I know, but you get used to it if you work with arrays long enough. Regardless of the strangeness of numbering in Java arrays, they do keep the elements in the array in the order they were added. That way we can always get back out the one we want. All we have to do is remember which one we want.

Now...lets combine arrays with the FractionChoices2.java sample from the Making Choices section. This means we need some way of checking more than one number. We could do it really poorly with a lot of code duplication...or we could do it with a loop.

Loops:

In programming, a loop is something that happens over and over again. So what you want to do is check, over and over again, each number in the numbers array. Java has a type of loop specifically designed to work with arrays. It is called a 'for loop.' It's called this because the keyword used to write it is 'for.' Line **10** in Code Snippet 5, part of the GroupLoop.java code sample, is the *for* loop declaration that works with the numbers array created on line **9**. Line **10** has the 5 distinct parts of all *for* loops.

```
9  double numbers[7] = {0.25, 11.003, -2.14, 0.14,
                                20001.3, -.000001, -10.0};

    1      2        3     4
10 for (int i = 0; i < 7; i++){
       .
       . 5
       .
19 }
```

Code Snippet 5

The standard for loop parts when working with any array are:

1. the *for* keyword,
2. an int to keep track of which array element we are working with, the loop's location tracker,
3. a check to see if the loop should continue,
4. an update that increases the location tracker by one, and
5. the code to execute every time we go around the loop, lines **11** though **18** in the GroupLoop.java example.

Line **10** reads, "create a *for* loop where i, the location tracking int, starts out as zero, the loop continues until i reaches seven, i goes up by one each time the loop goes around, and the code in the {} characters is executed each time the loop goes around."

Line **10** is a standard structure in Java. It is so standard that many code editors will generate most of it for you. By tradition, the location tracker is named 'i'. It stands for 'index.' No need to

memorize this little bit of information. It's just some programming culture trivia.

So let's put the group loop code all together now. We've looked at most of the parts before in Section 3. Line **10** is the only bit that's new.

```
1  public class GroupLoop {
2
3
4    public static void main(String[] args) throws Exception{
5       checkDoubles();
6    }
7
8    public static void checkDoubles() throws Exception{
9       double numbers = {0.25, 11.003, -2.14, 0.14,
                          20001.3, -.000001, -10.0};
10      for(int i = 0; i < 7; i++){
11        double aNumber = numbers[i];
12        if(aNumber > 0 && aNumber < 1
                  || aNumber < 0 && aNumber > -1){
13           System.out.printf("%f is only fractional.\n",
                                                  aNumber);
14        }
15        else{
16           System.out.printf("%f has a whole number
                                  component.\n", aNumber);
17        }
18      }
19   }
20 }
```

Code Sample - GroupLoop.java

Details:

Remember, i is used to keep track of which array element the code is using as we go through each of the values. In programming speak, going through each of the values of an array is called 'iterating over the array.' You will hear that phrase a lot in programming books and tutorials.

Line **10** reads, "let the double aNumber be the ith element in the numbers array" where ith can be the zeroth, oneth, twoth, threeth, fourth, fifth, or sixth element. Notice again I didn't use the word first. The rest of the code executed by the loop, lines **12** through **17**, come directly from the FractionChoices.java code sample from Section 3. If you don't understand those lines, look there.

Changing Arrays:

The value of an element in a Java array can be changed. For example, we could change the zeroth value of the numbers array to be 10.25. We reuse the [] characters to do this.

```
numbers[0] = 10.25
```

Code Snippet 6

Code Snippet 6 reads, "let the zeroth double of the numbers array be 10.25." Now we'll get 10.25 every time we get the zeroth element back out.

Remember back at the beginning of this section when I said that arrays were dangerous? I'm going to show you how right now. Take a look at the GroupBad.java code sample.

```
1 public class GroupBad {
2
3
4     public static void main(String[] args) throws Exception{
5        crashIt();
6     }
7     public static void crashIt(){
8        double numbers[7] = {0.25, 11.003, -2.14,
                              0.14, 20001.3, -.000001, -10.0};
9        numbers[7] = 45.9;
10    }
11 }
```

Code Sample - GroupBad.java

Details:

Line 9 is VERY bad. Even though the *numbers* array is seven elements long, there is no seventh element. Remember Java arrays start at zero so the sixth element is the last real element. In some other languages, like JavaScript, this wouldn't be a problem. JavaScript would add a new element to the end of the numbers array. We are not writing JavaScript. We are writing Java and Java doesn't have arrays that can change size. Once we said the array had seven elements it can never have more and can never have less.

The reason this code is so bad is your code will attempt to write a value into RAM outside of the array's location. It causes your application to exit. The Java Compiler isn't smart enough to catch this type of error in your code. You must always check your code yourself.

Java handles this problem we've created by exiting the application and saying why it is quitting. When you run the GroupBad application you'll get something weird that looks like this in your console. In techie terms, this is a 'call stack.'

```
1  Exception in thread "main"
2 java.lang.ArrayIndexOutOfBoundsException: 7
3   at GroupBad.crashIt(GroupBad.java:10)
4   at GroupBad.main(GroupBad.java:5)
```

Results- GroupBad Application

Call stacks tell you exactly what the problem was when your code ran and where the problem is. Line 1 of BadGroup's results tells us the problem, an Exception, has happened. Line 2 tells us the type of Exception. It reads, "You attempted to add something to the array, but the index you used was out of bounds. The location was too large. Oh, and by the way, the index you attempted to use was 7." That makes sense. We expected a problem when we used 7 as our location in the array. But what are lines 3 and 4?!

Line 3 tells us, "The problem happened in the GroupBad's crashIt procedure in the GroupBad.java file on line 10. Just this much information is often enough to know where to fix our mistake, but line 4 gives us more help. It says, "Oh...by the way...you called crashIt inside of GroupBad's main procedure in the GroupBad.java file on line 5." That's a lot of information but when you are trying to fix bugs, and you will need to, these call stacks are just as important to understand as any part of Java's syntax or grammar. Don't forget about them. You WILL see them often when you are writing your own code.

Other kinds of loops:

For loops are not the only kind of loop in Java. There is also a *while* loop. While loops are used in situations that are not linked directly to arrays. They are used in situations where an application experiences 'states'. In English there is a saying, "Work while the sun shines." This saying means that while the sun is shining you should work and then stop when night comes. That is exactly how a while loop in Java works. Code Snippet 7 shows some code of how this English saying could be written.

```
1 int sunIsShining = 1;

    1            2
2 while(sunIsShining == 1){
3
4   ..work, work, work. 3
5   if(currentTime >= sundown){
6         sunIsShining = 0;
7   }
8 }
```

Code Snippet 7

Line **2** shows how to declare a while loop. Unlike the *for* loop, *while* loops only have three parts, (**1**)the while keyword, (**2**)the check to see if the loop should continue, and (**3**)the code to execute each time we go around the loop.

One common way a while loop is used in Java is to do user IO. In the Remembering User's Info section the user didn't get to enter more than one pair of ints. A calculator, like that one, which shuts down after the first calculation is a BAD idea. Let's rethink that example. First, instead of multiplying lets divide two ints. Let's also create a procedure so we can get out of the main function as soon as possible. That is almost always a good idea....and we'll use what we know about printf and Scanners to do user IO. Computers can't divide by zero so we'll create that restriction. If the user enters 0 as the number to divide by, let's tell them 'Opps. There's a problem' and exit the app.

In this example I'm going to use a few math terms. Don't let them freak you out. Dividing 6 by 2 and getting 3 can be represented as $6 \div 2 = 3$. We can also think of it like a fraction and represent it as $6/3 = 2$. When talking about the 6 in this example we call it the **dividend**, the number to be divided. The 3 is called the **divisor**, the number to divide by, and the 2 is the **quotient**, the answer. Don't freak. You know this from long ago when you first learned how to do division. You just don't quite remember it. That's OK. If you get confused just look back at this 6-3-2 example.

All right. Lets do some dividing!

```
2 public class DivideLoop {
3
4
5   public static void main(String[] args) throws Exception{
6     divideInts();
7   }
8  public static void divideInts() throws Exception{
9     Scanner systemInScanner = new Scanner(System.in);
10    System.out.printf("enter an integer: ");
11    int dividend = systemInScanner.nextInt();
12    System.out.printf("What integer would you like to
                                    divide %d by?\n", dividend);
13    int divisor = systemInScanner.nextInt();
14    while(divisor != 0){
15      int quotient = dividend/divisor;
16      System.out.printf("%d / %d = %d\n", dividend,
                                              divisor, quotient);
17      System.out.printf("enter an integer: ");
18      dividend = systemInScanner.nextInt();
19      System.out.printf("What integer would you like to
                                    divide %d by?\n", dividend);
20      divisor = systemInScanner.nextInt();
21    }
22    System.out.printf("Oops. There's a problem. You can't
                                            divide by zero.\n");
    }
23 }
```

Code Sample - DivideLoop.java

Details:

The DivideLoop.java code sample is written non-optimally. There is code duplication going on. Lines **10** - **12** and **17** - **20**, our user IO code, are duplicate sets of lines. While there are Java ways to remove this duplication, they would have complicated the while loop example. Rather than expose you to those more advanced topics I chose to not confuse you. Instead, I chose to keep the sample simple. There are a whole bunch of examples of how to solve this specific duplication problem on the web so I'll leave it for you to explore later.

Line **14** contains the while loop and reads, "While the divisor is not zero execute the loop code." The code the loop executes repeatedly is on lines **15** to **20**. Notice that the divisor is entered by the user on line **13** so the while loop can check to see if it is zero or not.

Line **22** is executed ONLY if the user enters 0 at line **12** or line **19**. This happens because divisor is now 0 and the loop code only executes while the divisor is not 0. When the divisor is 0 the loop is all done, 'terminated' in techie speak.

So lets try some numbers. 20 / 5 = 4. Yep. That worked. -27 / 3 = -9. Yep. That worked. 12 / 5 = 2. What???? Shouldn't that be 2.4? Hmm.... Nope. It actually should be 2. "Why?" you ask. Because we told the computer, through our code, to only deal with integers. Remember, computers are stupid. They aren't smart like you. Java can't look at a situation and say, "Oh. There should be a fractional part to the result of this calculation. I'll add one on." Nope. It can't do it. If you want to have fractional parts to the answers you would need dividend, divisor, and quotient to be doubles. That way the computer will know you always want fractional parts.

But remember, computers are stupid. That means your user can't enter 6 and 3 to get 2. If you are using doubles they would enter 6 and 3 but would get 2.0. Try converting the code to use and display doubles as an extra bit of practice.

Printing Doubles

Representing Things

Lorem Ipsum

1. How can Java represent things we see in the real world and complex ideas?
2. What is a class?
3. What is an instance?
4. How is a class made and used?

The world around us is made up of things. We make up groups of things all the time. This thing fits in the bird, goose, and snow goose groups and many others. It could also fit in a 'things with white coloration' group.

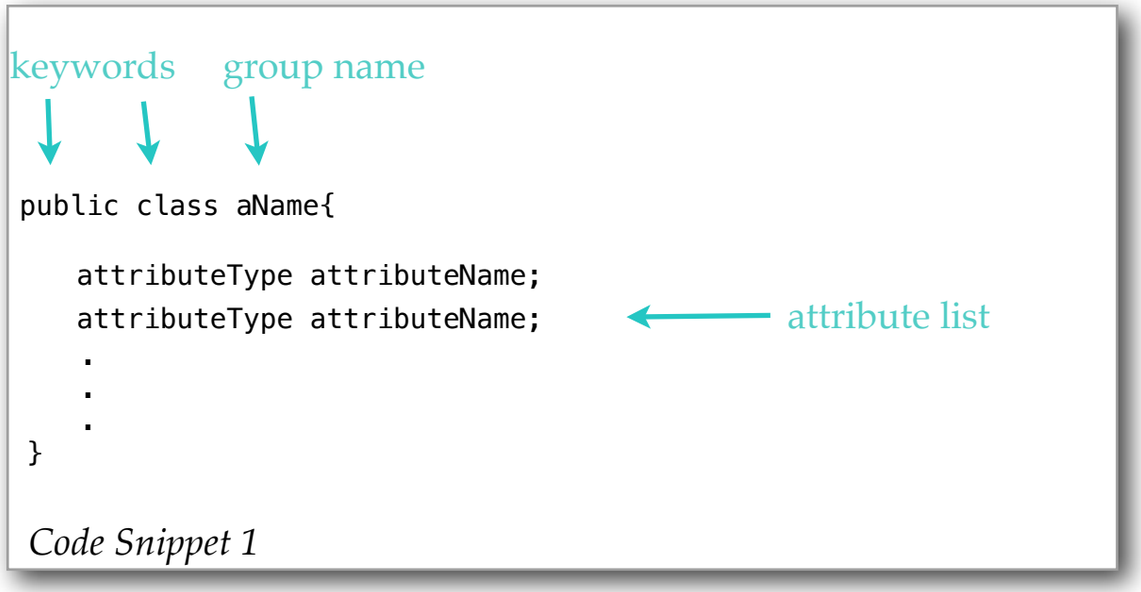
Arrays are nice, but how in the world can you represent something like a customer using arrays? A clothing customer could have a name, age, and inseam. That means for each customer would need to consist of a string, int, and double. Obviously we can't put these three types of things into one Java array because Java arrays can only hold things of the same type.

There are ways to use arrays to pull this of but we wold have to write really bad code. It would be hard to write, modify, and debug. So lets look at a better way.

Because humans think in things, there is a way to represent a thing in Java. It is called a class. In the earlier sections of this book we called them groups. Why? Because a class represents and describes a group.

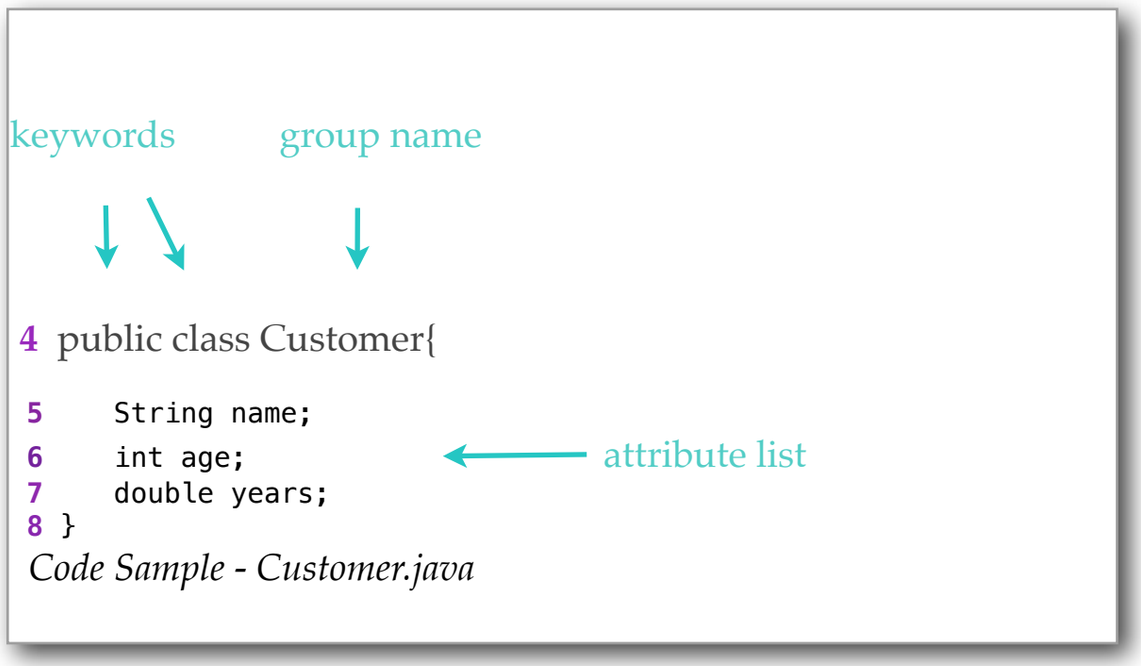
Because classes represent actual things we have to describe what the thing 'looks like.' We said that a clothing customer would need to have a name, age, and inseam. Why those? The only reason is because those pieces of information are all we need in our pretend situation. In other situations we may need hat size, sleeve length, or any number of other measurements. For our example let's keep it simple.

The pattern for describing something's group using a class is always the same. The *class* keyword is used to tell the computer to prepare for a group description and public means any other code in the application can use this class. Using public here has the same result as the public we have used with the functions and procedures in the earlier book sections. Both mean any code in the app can use them.



Code Snippet 1

The class keyword, as seen in Code Snippet 1, is always followed by the name of the group being described and then within the {} characters a list of types and names is used to describe each of the traits of the group. So lets stop talking about patterns and create an example customer instead.



Code Sample - Customer.java

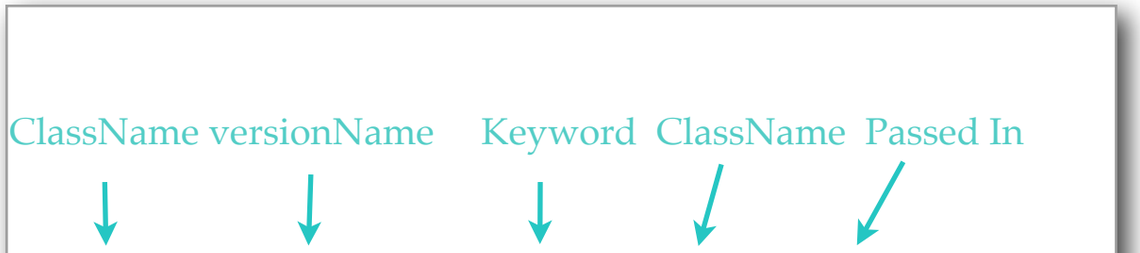
Details:

Line 4 of the Customer.java code sample has the *class* keyword and the name of the group of things we're describing, *Customer.* Line 5 looks a little strange, so we'll come back to that in a second. Line 6 is an int attribute (trait) of Customer named *age* and line 7 is a double attribute named *years,* meaning the number of years they have been a customer. Those last two seem familiar. They are like the values we've stored and used in the other code examples but here they are used to describe the Customer 'group.'

Now let's take a look at line 5. The type of this attribute of our Customer is 'String'. We haven't used this type before but it is how Java represents text and is an attribute called *name*. It can hold any amount of text. Notice that *String* starts with a capital letter like the *Customer* class does, but *int* and *double* don't. That's because String is one of Java's many standard classes. Unknowingly you've used other standard Java classes (Scanner and System). Remember, we called them groups.

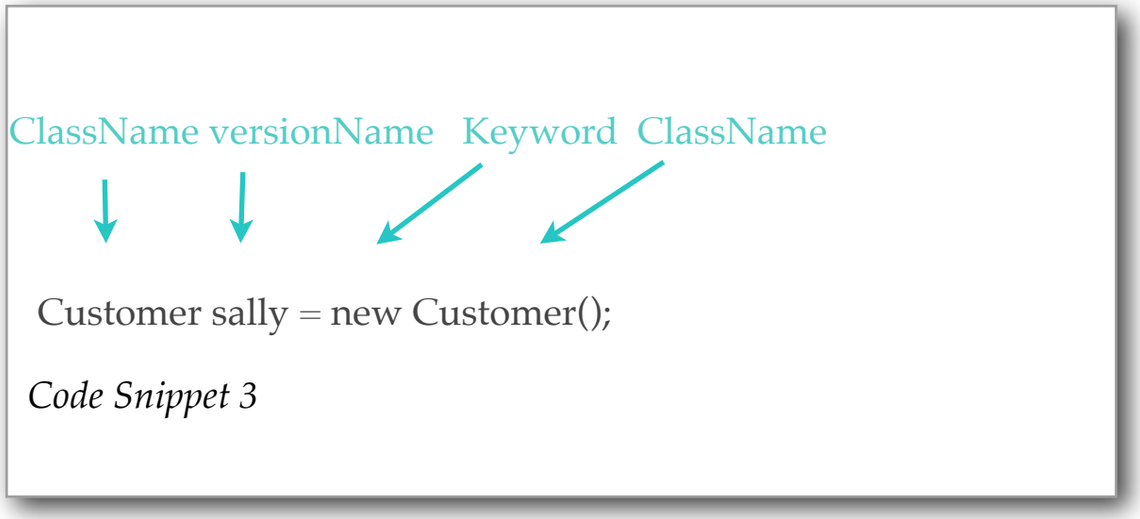
So there is our description. Each version of Customer we create will have three attributes (traits); a name represented by a String, an age represented by an int, and the number of years they have been a customer represented by a double.

Back in Section 2 we created a version of the Scanner class (group). It is shown again in Code Snippet 2 and has all of the parts used to create new versions of Scanner.



Code Snippet 2

I know. I know. That's a lot of stuff. But you've seen it before so only the *class* idea is new here. "But how do we create one of our Customers?" you ask. Almost exactly the same way.



Code Snippet 3

We don't need to pass stuff in, so we don't. That is different than when we used Scanner. The people who wrote the Scanner code decided it would need to have System.in or some other similar thing passed in. That was their choice. We have chosen not to have anything passed in because we don't need to.

Code snippet 4 shows the rest of the pattern to follow; First a version of Customer is created. The first line of the code snippet reads, "Create a Customer and identify it as cheapCustomer."

```
Customer cheapCustomer = new Customer();

cheapCustomer.name = "sally";

cheapCustomer.age = 45;

cheapCustomer.years = 5.25;
```

Code Snippet 4

After a Customer version is created its attributes (traits) can be set. This is done by listing the identifier, *cheapCustomer*, a . character, and then the identifier for the attribute. The second line of the code snippet reads, "Let cheapCustomer's name be 'sally'." The age is set on the next line. It reads, "Let cheapCustomer's age be 45." Get it? Let's do the last line just to make sure. It reads, "Let cheapCustomer's years be 5.25."

There we go. We've created a customer and set the customer's attributes. Great. Let's see how to do this as part of an app. Keeping it simple, the app will create a Customer and scan to get the values for the name, age, and years from the user. Then it will print out a description of the Customer. Just like in previous examples, this will be done in a procedure to get us out of the main method as quickly as possible. We will also put main in its own class, SillyCustomerRunner. This will help us later. Just trust me for now. Coincidentally, this choice also makes it possible to show you how versions of one class can interact with another.

```
3  public class SillyCustomerRunner {
4
5   public static void buildACustomer(){
6        Scanner systemInScanner = new Scanner(System.in);
7        System.out.printf("Enter the customer's name\n");
8        String aName = systemInScanner.nextLine();
9        System.out.printf("Enter %s's age\n", aName);
10       int anAge = systemInScanner.nextInt();
11       System.out.printf("How many years has %s been a
                                              customer?\n", aName);
12       double someYears = systemInScanner.nextDouble();

13       Customer aCustomer = new Customer();
14       aCustomer.name = aName;
15       aCustomer.age = anAge;
16       aCustomer.years = someYears;

17       System.out.printf("%s has been a customer for %f years
                                    and is %d years old\n",
                                  aCustomer.name, aCustomer.years,
                                                  aCustomer.age);
18   }
19   public static void main(String[] args) throws Exception{
20       buildACustomer();
21   }
22 }
```

Code Sample - SillyCustomerRunner.java

Details:

We've seen stuff like lines **6** - **12** of the SilyCustomerRunner.java code sample before. Remember in the previous sections of this book where we needed information from the user? The major difference in SillyCustomerRunner.java is at line **8** where the *nextLine* function of the Scanner class is used. *nextLine* is what you

use when you need to read in a String. Why isn't this nextString so it matches nextInt and nextDouble? The old guys with white beards decided to break the pattern. Yep, your right. That was silly.

Lines **13** - **16** we've already described so no need to duplicate that description here. Line **17** is where we see how to print out a double. We've only dealt with Strings and ints before. When you want to print out a double use the %f indicator. %f stands for *floating point*. %f was chosen since doubles have a decimal point that can 'float around.' We can see this if we look at a few numbers.

- 3.2
- .000000001
- 134567.00000000000007

The decimal is never a fixed distance from the beginning or end of the number. It floats around. In 3.2 it is in the second spot, in .000000001 it is in the first spot, and in 134567.00000000000007 it is in the seventh spot. So you see, the spot where the decimal is isn't fixed. Yes is it always between the ones and tenths places, but that isn't what we're talking about here. We're talking about how far the decimal is from the beginning of the number.

Storing Things:

But entering data and printing it out is useless if the data can't be stored. Section 2 is about storing and retrieving regular data types like ints. Now we'll figure out how to do it with instances, versions, of classes.

You can't just pass an instance of a class to the PrintWriter's *printf* function. The values of the attributes in the instance have to be pulled out, similar to when we used printf to show the user information in the SillyCustomerRunnable.java code sample.

In the CustomerCreatorAndWriter.java code sample, five customers are created and stored in the *customer.data* file. Nearly all of what happens in this sample is similar to the samples from this and previous sections of the book. If you are getting confused, PLEASE go back and review the part of the book explaining the thing confusing you. Being a 'good soldier' and trudging forward will not be helpful to you. Instead, go back, regroup, refresh your understanding of the previous examples, and then come back here to expand on what you know.

```
4  public class CustomerCreatorAndWriter {
5     static Customer[] customers = new Customer[5];
6   public static void buildCustomers() throws Exception{
7         for(int i = 0; i < 5; i++){
8              Scanner systemInScanner = new Scanner(System.in);
9              System.out.printf("Enter a customer's name\n");
10             String aName = systemInScanner.nextLine();
11             System.out.printf("Enter %s's age\n", aName);
12             int anAge = systemInScanner.nextInt();
13             System.out.printf("How many years has %s been a
                                              customer?\n", aName);
14             double someYears = systemInScanner.nextDouble();
15
16             Customer aCustomer = new Customer();
17             aCustomer.name = aName;
18             aCustomer.age = anAge;
19             aCustomer.years = someYears;
20
21            CustomerCreatorAndWriter.customers[i] = aCustomer;
22        }
23  }
24    public static void saveCustomers() throws Exception{
25        File customersFile = new File("customer.data");
26        PrintWriter customerWriter =
                               new PrintWriter(customersFile);
27        for(int i = 0; i < 5; i++){
28            Customer aCustomerToPrint =
                        CustomerCreatorAndWriter. customers[i];
29            customerWriter.printf("%d %f %s\n",
                    aCustomerToPrint.age, aCustomerToPrint.years,
                                             aCustomerToPrint.name);
30            customerWriter.flush();
31        }
32    }
33    public static void main(String[] args) throws Exception{
34       CustomerCreatorAndWriter.buildCustomers();
35        CustomerCreatorAndWriter.saveCustomers();
36    }
37  }
```

Code Sample - CustomerCreatorAndWriter.java

Details:

The CustomerCreatorAndWriter.java code sample contains three procedures, *main*, *buildCustomers*, and *saveCustomers*. The *buildCustomers* and *saveCustomers* procedures were named to reflect what they do, as discussed in Section 1. The code in *buildCustomers* is nearly the same as the part of the SillyCustomerRunner.java code sample we just looked at. *buildCustomers* asks the user the same questions and then creates an instance (version) of Customer using the information the user supplied. A *for* loop is included so the user can create five Customers and these are stored in an array of Customers. Information about and examples of loops and arrays can be found in Section 4.

The array of Customers is created on line **5** of the example. It looks different than what you've seen before. It reads, "Create an array of five Customers and store it in an attribute of CustomerCreatorAndWriter named 'customers'. We'll add customers to it later." But what does *static* mean at the beginning of the line?? *static*, regardless of if it is in front of a function, procedure, or attribute always means the same thing--it is part of the class. That means there is no need to create an instance of the class to use it. You can see we never create an instance of CustomerCreatorAndWriter. We don't need to. But we do create instances of Customer since we need five of them. There is much more to *static* than this, but this much understanding is sufficient for now. Feel free to look it up online.

The *saveCustomers* procedure on line **24** combines the information from Section 2 about storing information and line **17** from the SillyCustomerRunnable.java code sample. When printf is used on line **29**, three % indicators are used--%d and %f, as in previous samples, and %s used to write out strings. That's it. Things can

now be stored for later use. Let's look at how to get them back out of the file.

Retrieving Stored Things:

Reading the data back in also uses the concepts described in the previous sections so the CustomerReader.java code sample should also look familiar. It consists of a procedure called readAndPrintCustomers. five customers were written out, so this example expects five customers to be read back in.

```
3  public class CustomerReader {
4   static Customer[] customers = new Customer[5];
5   public static void readCustomers() throws Exception{
6       File customersFile = new File("customer.data");
7       Scanner customerScanner = new Scanner(customersFile);
8       for(int i = 0; i < 5; i++){
9             int anAge = customerScanner.nextInt();
10            double someYears = customerScanner.nextDouble();
11            String aName = customerScanner.nextLine();
12
13            Customer aCustomer = new Customer();
14            aCustomer.name = aName;
15            aCustomer.age = anAge;
16            aCustomer.years = someYears;
17
18            customers[i] = aCustomer;
19       }
20 }
21   public static void displayCustomers() throws Exception{
22       for(int i = 0; i < 5; i++){
23           Customer aCustomerToPrint = customers[i];
24           System.out.printf("%s has been a customer for %f
                                    years and is %d years old\n",
                                          aCustomerToPrint.name,
                                          aCustomerToPrint.years,
                                          aCustomerToPrint.age);
25       }
26   }
27   public static void main(String[] args) throws Exception{
28       readCustomers();
29       displayCustomers();
30   }
31 }
```

Code Sample - CustomerReader.java

Details:

Line 8 starts the loop to read in the five Customers. Lines 9, 10, and 11 use the nextInt, nextDouble, and nextLine discussed earlier in this section to get the information for each of the Customers stored in the file. 13 - 16 are the same as when we created a Customer and

stored its attributes as part of the SillyCustomersRunner.java code sample. When the customers were written out to *customers.data* each customer's information was placed on a single line. This means that there are five lines of data in the file, each representing a single customer.

Each of the Customers is added to the customers array attribute of the CustomerReader class on line **18** as the code goes around the loop. The customers array is used again in the displayCustomers procedure to print out each of the five Customers that were read back in.

There it is. Stored and read back in. Whew. That was a lot of stuff.

Where Next?

A good start is only a start. If you don't keep going did you really start or are you right where you began?

There are still many things about Java you don't know. A few of these are listed here with links to help in your discovery process. Also, a good tutorial would be a great idea.

Here are some links that might be of help as you continue your exploration.

Java Constructors.

Instance methods.

More information about *static.*

Browse through the Java API to see the huge amount of other things that are possible, talk with others new to Java about what you've learned, and play with the code. It will help you build on what you know.