Learn By Doing Quick Guide to Java Programming

Table of Contents

Chapter 1: Introduction

Section 1.1: Objective

By the end of this book it is my hope that the reader will have a good understanding of what a programming language is, and how they are used. More importantly, the reader will gain sufficient knowledge of the Java programming language, such that they can write reasonably complicated programs. I believe that the best way to learn is by doing. This is why this is a project based book where the reader will be walked through writing various programs and encouraged to write others on their own. Downloadable versions of the example problems and the solutions to the practice problems can be found on the website at www.jtjackson.org. Please note that copying and pasting the programming examples from this book could result in issues when attempting to run them. This is why I've also placed the programs on the website. This guide is not meant to be a comprehensive overview of the java programming language because it is a huge language that whole textbooks have been dedicated to. This book is meant to give the reader a great foundation to not only write powerful programs but to also be able to progressively build more knowledge through practice and other resources.

Section 1.2: Who is this for

This book is meant to provide a foundation in the java programming language. With that being said, this book is for both the beginning programmer with no experience, and the experienced programmer who would like to quickly learn java. This book should be an easy read for both the novice programmer and the experienced programmer. For the experienced programmer reading this, I apologize when I impart knowledge that seems rudimentary to you but please bear with me. I try to disperse the elementary knowledge throughout the book so as not to make you feel as though you had to suffer through an introductory Computer Science class before getting to the good stuff. I truly hope that you enjoy the book and would love your feedback which can also be provided on the website.

Section 1.3: What is a programming language

A programming language is a language that is used to write instructions for a computer. There are many programming languages out there and new ones are being developed every day. Programming languages have some things in common with human languages. The same way that each human language has certain syntax rules that have to be followed in order to make a valid sentence, a programming language has syntax rules that must be followed in order to make a valid program. A program is a valid set of instructions in a particular programming language. A program will make a computer behave a certain way. The language that we'll be focusing on in this book is Java. It is a high level, object oriented language. High level means that it is a language that is close enough to the semantics of a human language (such as English or Spanish) that it is easily interpreted by humans. High level languages do not run directly on the hardware of a computer. Instead they are compiled or interpreted into progressively lower level languages by other programs (Compilers and Interpreters) until they become binary code (the 0's and 1's that run directly on the hardware). An object oriented language is a language that allows the programmer to create constructs in the language that represent real world objects. We'll talk more about the creation of objects later.

Section 1.4: Setup

In order to run java programs you must first download the java development kit (JDK) onto your computer, if you don't already have it. Follow this link http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html and select the java development kit for your particular computer platform. There are several versions for each operating system (windows, Linux, etc.). The 32 bit processor version may be represented using x86 and the 64 bit processor version may be represented using x64. If you're not sure which processor your computer uses, a simple google search can be used to help you figure it out. In particular searching the phrase "determine if your computer is 32-bit or 64-bit" produces some relevant results. Download the JDK. During the process, if the folder that the kit will be stored in is mentioned take note of it. You will need it later. I'm a windows user myself and so I can't give directions on how to set the command line/prompt up with java for other operating systems but a google search should render it a pretty simple task. If you're a Linux user, the following link may be helpful.

https://access.redhat.com/documentation/en-US/Red_Hat_JBoss_Portal/5.1/html/Installation_Guide/Pre_Requisites-Configuring_Your_Java_Environment.html

Please note that instead of java 6 you should be using version 7 or 8. The following link may be useful in adding the java directory to the path for Linux and Mac users.

http://cloudlink.soasta.com/t5/CloudTest-Knowledge-Base/Adding-JDK-Path-in-Mac-OS-X-Linux-or-Windows/ta-p/43867.

Below are directions for setting up java on the windows command prompt.

1. First you must determine where your java development kit folder is located. If you didn't before, you'll need to find it now. It will likely be located in the C:\Program Files\Java

   folder. The folder will be named something like jdk1.8.0_66, possibly with a different version number following JDK. Open this folder then open the bin folder within it. Keep this folder open so that you can copy the path later

2. Now open the start menu and if Control Panel isn't one of the immediate options, type it into the search bar and select it
3. Select the system folder
4. Next select Advanced System Settings
5. Then select environment variables
6. Under the system variables heading scroll down to the path variable, select it, and click the edit button
7. If the end of the path doesn't already include a semi-colon (;) add one. Then copy the path to the JDK folder to the end of the contents of the path variable and add \bin to the end if it isn't already there. Click Ok three times.
8. Now to test this Open the start menu and then open the command prompt. If you already had one open, close it and open a new one. Now type java -version. If you get an error message instead of some kind of information about the java

version then you may have to trouble shoot your efforts. Go back and make sure you used the correct path for the JDK bin folder. If that doesn’t work, use google. It’s likely that others have encountered the same problem that you’re having now.

Chapter 2: Your first program

Section 2.1: Writing the Program

Now it's time to write your first program. Open a text editor. So that there aren't any issues with the program it is preferable that you use a plain text, text editor. That is, a text editor that doesn't have formatting (so not Microsoft Word). If you're using a windows computer, you can use notepad. Otherwise use some other plain text editor. If your computer doesn't have a plain text editor, you can download Notepadd++ by googling the term "Download Notepad++".It may be a good idea to create a folder for our projects somewhere on your computer and add a folder for each chapter where we create new programs. Then create a folder for each chapter section where a new program is created and finally create a folder for each example program. This is necessary because no folder can have two files with the same name and we create different versions of the same program at times. Open the text editor and save the current empty file as *HelloWorld.java* in a folder called *2.1.1* (the example program number). All java files must end in the .java extension. Now it's time to create the shell of the program. Type the following and save the program.

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

Prog. 2.1.1a

This may seem confusing. You don't have to understand what any of this means now but rest assured that all will make sense later. The only thing that you have to know is that the name of the class (in this case *HelloWorld*) and the name of the java file must match. This includes the case. Java is a case sensitive language. *HelloWorld* and *helloworld* would be two different class names. Think of this as the shell of the program. For now, all of our programs will have a similar setup, with only the class name varying. Within the second set of curly braces (the main block) add the following changes so that the program looks like this.

```
public class HelloWorld {
    public static void main(String[] args) {
            System.out.println("Hello World");
    }
}
```

Prog. 2.1.1b

The change you have just made is called a print statement. We will be using this repeatedly throughout this book. Now it is time to run the program.

Section 2.2: Compiling the Program

Before a java program can be run it must first be compiled. Compiling the program is what converts the java program to machine code that can be run on the computer. Compiling a java program is simple. Enter the following into the command line

javac HelloWorld.java

If you get an error message, check your program. Check that both sets of curly braces were used. Also check that the class name matches the file name. Look through the program and ensure that everything has the proper case. The spaces/tabs aren't important but otherwise it is important to make sure that both programs match each other. The error message should give you some idea about what the issue is. Debug the program until it can be compiled without errors. If you can't figure out what the issue is, google the error message. I've yet to encounter an error message that isn't addressed on the internet somewhere. If you still can't figure it out, feel free to visit my website (www.jtjackson.org) and discuss it with others on the discussion board or get my contact information and contact me directly. I'll also periodically check out the discussion board to answer unanswered questions.

Section 2.3: Running the Program

Now that the program has been compiled successfully, it is time to run it. Enter the following onto the command line.

java HelloWorld

Your program should print out the message Hello World. Congratulations! You have written your first java program. You can play around with the program, changing the message within the double quotes to print anything you want. Note that as long as a program hasn't been changed, it doesn't have to be recompiled to be run again.

Chapter 3: Comments

Comments are an important part of programming. A comment is a line or block of code that is only meant for the programmer. The compiler will skip over comments. They are used to document what is going on in the code. Seeing that so far you've only been introduced to print statements, they may seem unnecessary, but they become much more important as your programs become more complicated. If you were to finish writing a program today and go back and look at it in 6 months (or maybe even 6 weeks), there's a good chance you won't remember what certain parts of the code were meant to accomplish. This is where comments come in. There are two types of comments in the java language. There are single line comments and multi-line comments. A single line comment always starts with two forward slashes. For example

//Hello World program

Ex. 3.1

A multi-line comment starts with a forward slash and an asterisk (/*) and ends with an asterisk and forward slash (*/). For example

/*This is

An example of

A multi-line comment */

Ex. 3.2

Of course a multi-line comment could be used as a single line comment as well. It just depends on what suits your particular needs at the time.

Proper documentation of code is very important in programming. You may not be the only person reading your code. Even if you can understand your logic 6 months from now, that doesn't mean someone else can. Plus it's just a good habit to get into.

Chapter 4: Variables, Expressions, and Various Print Statements

Just like any other programming language, java allows a user to create variables. A variable is used to stored data for you to access/use later. Java has several different types of variables, *String*, *int*, *char*, *boolean*, *float*, *double*, *byte*, *short*, and *long*.

Section 4.1: Types

Please note the following things. Saying that an integer is signed means that it can be negative or positive. An unsigned data type would be an integer data type that could only represent positive values. The number of bits of a data type is the amount of memory available for a value of this type. The amount of storage directly corresponds to the range of values that the type can store.

String

A *String* is just what it sounds like. A *String* of 0 or more characters represented between a pair of *double* quotes.

Short

A *short* is a 16 bit signed integer. A short has a minimum value of -32,768 (-2^{15}) and a maximum value of 32,767 (2^{15}-1).

Int

An *int* is a 32 bit signed integer. It has a minimum value of -2^{31} and a maximum value of 2^{31}-1.

Long

A *long* is a 64 bit signed integer. It has a minimum value of -2^{63} and a maximum value of 2^{63}-1.

Float

A *float* is a single precision decimal number.

Double

A *double* is a *double* precision decimal number.

Boolean

A *boolean* has two possible values, true or false.

Char

A *char* is a single character represented between a pair of single quotes.

The data types we'll be focusing on are *String*, *int*, *char*, *Boolean*, and *double*.

Section 4.2: Intro to Declaring, Initializing, and Using Variables

In order to create a variable it must first be declared. For example if you wanted to declare a *String* variable called name, the following code would be used.

String name;

The *String* type is the only built-in data type that uses a capital letter. Once a variable is declared it must be initialized (given a value) before it can be used. For example, to give the name variable the value "Bob", the following code would be used.

name = "Bob";

A variable cannot be initialized before it is declared but it can be declared and initialized at the same time. The following code would be used to declare the name variable and initialize it to the String "Bob".

String name = "Bob";

It is important not to forget the semi-colon (;) at the end or errors will occur. When naming your variables keep the rules in mind. A variable can be any sequence of letters, numbers, dollar signs and underscores that do not begin with a number and includes no spaces. Variable names are case sensitive so a variable called *name* would be different that a variable called *Name*. Keywords (or reserved words) are words that have special meaning in a programming language. Variable names cannot be a keyword. For example you cannot name your variable *public*, *class*, *int*, *char*, and so on. Now let's try accomplishing the same goal a little differently. Copy your *HelloWorld* program into a folder called *4.2.1*. In the new *HelloWorld* program, replace the print statement within the main block of your program with the following code.

String hello = "Hello World";

System.out.println(hello);

Prog. 4.2.1

Re-compile and run the program. The program should still print out Hello World. Just like before, you can test this program out with a variety of different strings. Now create a new program called vars.java and add the program shell that I described above with the class name *vars*. Add the following code in the main block.

String name = "Bob";

String greeting = "Hello";

System.out.println(greeting+" "+name);

Prog. 4.2.2

Compile and run the program. The program should print out Hello Bob. So what did we just do here? We concatenated (combined) the *greeting* string with the string that consists of a single space, and the *name* string and printed the new resulting string. In java, two or more strings can be combined using the plus operator. It doesn't matter if the strings are combined directly in the print statement or if the strings are concatenated in another string

and then printed using the print statement. The same task could have just as easily been concatenated and printed using the following code

String name = "Bob";

String greeting = "Hello";

String greet_person = greeting + " " + name;

System.out.println(greet_person);

Ex. 4.2.1

Or with

String greeting = "Hello Bob"

System.out.println(greeting);

Ex. 4.2.2

There are infinite possibilities. Play around with it and get an idea of what works and what doesn't. What about integer variables? An integer variable would be declared as following

int i;

Then it would be initialized as

i = 0;

Note that an integer variable can be initialized to any integer within the valid range mentioned above, just like a string variable can be initialized to any string. It could also be declared and initialized at once as follows.

int i = 0;

Simple integer values can be stored in an integer variable, as you've just seen, or an expression could be stored in an integer variable.

Section 4.3: Integer Expressions

Below are a few examples of integer expressions.

i=4+3;

i=6*7;

i=8/2;

i=9-5;

i=16%3;

Ex. 4.3.1

An arithmetic expression can be as complicated as you would like. It can also include parentheses to shift precedence (the order of operations). I'm pretty sure at least the first 4 out of 5 of those operators don't require my explanation. There's addition, multiplication, division, and subtraction. We'll just take a quick look at the fifth expression. The percent sign (%) is the remainder operator in java. The result of the expression would be 1 because 16 divided by 3 leaves a remainder of 1. One more important thing to note is that expressions stored within integer variables will explicitly have an integer result. If you were to store the expression 20/6 in an integer variable, the integer portion of the result would be stored in the variable and the decimal portion would be dropped. So the result of 20/6 would be 3. As a side note, although we've so far only defined an integer expression as those that require some calculation, a single integer is an integer expression (i.e. 20). It is in fact the simplest form of an integer expression.

The print statement doesn't only print strings. It also prints integers and all of the other built in data types. Behind the scenes, it converts these types to strings before printing them to the screen. Let's see some examples below. Copy your *vars* program to a folder called *4.3.1* and remove the code within the main block. Then replace it with the following code.

int x=5

int y = 15;

int z = y/x;

System.out.println("”+y+”/”+x+”=”+z);

System.out.println("13*2=”+13*2);

Prog. 4.3.1

Compile and run the program to see the results. So what just happened here? This simple program just demonstrated several concepts of the language. First, note the initialization of *z*. This demonstrates that an expression can optionally include variables. The value stored in *z*, as demonstrated before, is the result of the expression assigned to it. Now look at the first print statement. There seems to be a lot happening with this line of code but it is actually very simple. Let's look at it piece by piece. The first two quotes represent a string just like before. But unlike before, there are no characters in between the quotes (not

even a space). That's because we are seeking to represent the empty string. That is a string with no characters. Now look at the *+y* following the empty string. We've already learned that a string can be concatenated with another string to create a new string, using the plus operator. In fact, any built-in data type can be concatenated with a string to create a new string. Behind the scenes, the java virtual machine (software that runs the java programs after they're compiled to intermediate byte code) will decide what the string representation of a value should be, depending on its variable type, and create a new string that is the old string concatenated with this representation. In the case of integers, the string representation of an integer value is simply that integer between quotes. The same thing happens more or less with characters, *longs*, *shorts*, and *booleans*. So now let's move on to the "/". This is just the string representation of the forward slash character being concatenated to the string before it. In this case, the string before it is the string representation of the value of the *y* variable. Then there's the *+x*. The same thing is happening here as with the *+y*. The string representation of the value of the x integer is being appended to the end of the string before it. After that the string representation of the equal sign is appended to the string before it. Finally the string representation of the value of the z variable is appended to the end of the string before it. That leaves us with the string *15/5=3* being printed.

The next print statement is much simpler but demonstrates yet another concept. As you can see the string 13*2= is being printed. That is simple enough. But concatenated to that string is the string representation of the value 13*2. Variables aren't the only things that can be concatenated to strings. You can choose to skip the middle man (or woman!) and concatenate the expression to the string directly. The following string would be printed to the screen.

13*2=26

It's important to note that an attempt to divide an integer by zero will result in an error when attempting to run your program (run time error). So before we finish with integer variables let's talk about operator precedence.

Section 4.4: Order of Operations

The order of operations for mathematical expressions follows the normal mathematical conventions. Any expression within parentheses will be evaluated first. After that, all multiplication (*), division (*), and remainder (%) operations will be evaluated from left to right. Finally all addition (+) and subtraction (-) operations will be evaluated from left to right.

Section 4.5: Double Variables

As you know, a *double* value is a decimal value. A *double* variable would be initialized and declared as follows

double d;

d = 4.0; or

double d = 4.0;

Ex. 4.5.1

Just like with integer variables, a *double* variable can have a simple decimal stored in it or it can have an expression stored in it. For example

d = 4.0 * 2.0;

The result would be 8.0. An integer can be used to calculate a *double*. Internally, the java virtual machine will just convert the integer value to a *double* value. For example

d = 4.0 * 2;

The result would still be 8.0. On the other hand, a *double* value cannot be used in an integer expression. An integer can even be stored in a *double* but it will internally be converted to a decimal. Otherwise, a *double* follows many of the same rules as an integer including the order of operations rules. As mentioned before, a print statement can also print *double*s. Play around with various expressions so that you can get a feel for how this works.

Section 4.6: Char variables

Speaking simplistically, a *char* variable is simply a string of length one that is represented using single quotes instead of *double* quotes. A *char* variable would be declared and initialized as follows.

char c = 't';

The character data type has some interesting properties. Internally, computers represent characters using integer values. Many systems use the ASCII format and this is how java chooses to interpret character values. For this reason characters can be used in arithmetic expressions. For example

int x = 'c' * 2;

Since the character lowercase c is represented in the ASCII system as integer value 99, the result would be 198. This property can also be used to convert a character representation of an integer value to an actual integer value. The characters '0' through '9' are represented in the ASCII table in order, with increasing values. See the very much reduced ASCII table below.

ASCII Decimal Value	Character
48	'0'
49	'1'
50	'2'
51	'3'
52	'4'
53	'5'
54	'6'
55	'7'
56	'8'
57	'9'

Figure 4.6.1

So in order to take a character representation of an integer and convert it to the actual integer itself, one must only do the following

int t = '8' – '0';

The result will be 8. This works because all of the integer variables are stored with sequentially increasing decimal values. So internally '8' – '0' is equivalent to 56 – 48 which has a result of 8. Subtracting the character representation of zero from any character representation of an integer will have a result of that integer's value.

A similar trick can be used to convert lowercase letters to uppercase letters and vice versa. As you saw when we multiplied the character 'c' by two and stored it into an integer variable, the result was automatically converted to an integer, whose value was that of the ASCII value of 'c' multiplied by two. On the other hand, if you were to perform arithmetic on a literal character and store it in a character variable, the value would automatically be that of the character whose ASCII value represents the result. The system performs these conversions behind the scenes. Before seeing an example, check out another reduced version of the ASCII table.

ASCII Decimal Value	Character
65	'A'
66	'B'
90	'Z'
97	'a'
98	'b'
122	'z'

Figure 4.6.2

I skipped many characters for the sake of saving space but as you can see, the uppercase alphabet characters are stored as values 65 through 90 and the lowercase values are stored as 97 through 122. The uppercase letters are in order and so are the lowercase letters. This means that the decimal representation of a lowercase letter is that of its uppercase letter – 32. So an uppercase letter A could be converted to lowercase and a lowercase letter could be converted to uppercase as follows.

```
char a;
a = 'A' + 32;
System.out.println(a);
a = 'a' – 32;
System.out.println(a);
```

Prog. 4.6.1

Java is funny about the resulting data type when you perform these kinds of conversions. If you were to use a character variable in the arithmetic instead of a literal character, you

would encounter an error trying to store the result in a character variable because the result would automatically be an integer.

The print statements are purely for the purpose of seeing the result of your code. Copy your *vars* program to a folder called *4.6.1* and replace the code in the main block with the code above.

Section 4.7: Boolean Variables

As mentioned above, a *boolean* variable as two different possible values. It can either be true or false. A *boolean* variable, named b, could be declared and initialized as follows.

boolean b = false;

Boolean variables can also have expressions stored in them. You'll see *boolean* expressions below but it is important to note that true and false are themselves, expressions. They are the simplest form of a *boolean* expression. Copy your *vars* program to a folder called *4.7.1* and replace the code in the main block with the following code.

```
int i = 1;
int x = 2;
boolean b = x>i;
boolean t = false;
System.out.println(“”+x+“>”+i+”: ”+b);
b=x>=i;
System.out.println(“”+x+“>=”+i+”: ”+b);
b = x<i;
System.out.println(“”+x+“<”+i+”: ”+b);
b = x<=i;
System.out.println(“”+x+“<=”+i+”: ”+b);
b = x==i;
System.out.println(“”+x+“=”+i+”: ”+b);
  b=x!=i;
  System.out.println(“”+x+ “!=”+i+“: ”+b);
  b = !t;
  System.out.println(“!t: ”+b);
  b = t;
  System.out.println(“t: ”+t);
```

Prog. 4.7.1

Take a look at the first expression, *x>i*. As you're probably already aware, this is the greater than comparison, which in this case checks whether *x* is greater than *i* and the result will be true. The greater than or equal to (>=) operator checks whether the first expression is greater than or equal to the second. The < operator checks whether the first expression is less than the second. The <= checks whether the first expression is less than or equal to the second. The == operator checks whether two expressions are equal. Notice

that we use two equal signs instead of one. This is because a single equal sign is used as the assignment operator, when storing a value in a variable, so java chooses to use a separate operator to determine equality. The expression *x!=i*, determines if *x* is not equal to *i*. If x is not equal to *i*, the result will be true. Otherwise, it will be false. The second to last expression, *!t*, is a negation expression. It means, not *t*. So if *t* is true, not *t* will be equal to false. If *t* is false, not t will be equal to true.

Like with any other expression, a *boolean* expression could become more complicated. More than one value can be evaluated within a *boolean* expression. For example, you could do something like this

int x = 5; int y = 6;

boolean check = true;

boolean result = (y>x) && (check==true);

boolean second_result = (y>x) || (check==true);

Ex. 4.7.1

Notice the *double* ampersand (&&) operator. This is the *AND* operator. If both expressions evaluate to true then the final result will be true. Otherwise the result will be false. Now notice the || operator. This is the *OR* operator. If either expression evaluates to true, the result will be true. Now notice the *check==true* portion of the last two lines of code. Saying *check*==true is actually repetitive and unnecessary. Since the variable *check* is an expression in and of itself, then saying check==true is not necessary. If check were true then saying check==true would return true and if check was equal to false then saying check==true would return false. In other words, doing an equality comparison on a *boolean* value would just return the same value. It's simpler to just use the value itself rather than forcing a comparison. Test it out if you're unconvinced. The last two lines of code can be replaced with the following code.

boolean result = (y>x) && check;

boolean second_result = (y>x) || check;

Now add the following line for the purpose of seeing the result and run and compile the code.

System.out.println("result: "+result+" second_result "+second_result);

Section 4.8: Shorts, Longs, Bytes, and Floats

A *short* is just a smaller version of an integer and would be declared and initialized using the following syntax.

short s = 0;

A *long* is just a larger version of an integer. Notice that a lowercase l is appended to the end of the integer value when using *long*s in the example below.

long l = 0l;

You can think of a byte as an even smaller version of an integer than a *short*. It is declared and initialized as follows

byte b = 4;

A float is a smaller version of a *double* and is declared and initialized as follows;

float f = 4.0f;

Notice that a lowercase f is appended to the end of the decimal when using floats.

Section 4.9: Constants

Constants are exactly what they sound like. Once they have been initialized, there value cannot be changed. This topic is so simple that it barely deserves its own section but it diverged from the last section topic enough to warrant the distinction. A constant would be declared using the syntax

final datatype name = value; or

final datatype name; name=value;

For Example

final int weight = 2;

As a side note on variable declarations and initializations, they can be done on the same line, as seen above. In fact, any number of statements can be done on the same line but people tend to put them on separate likes for the sake of clarity.

Section 4.10: Static Casting

Sometimes it is necessary to convert from one variable type to another. This is called casting. This can only be done in some cases. For example, you cannot cast a *boolean* value to an integer. On the other hand, since a character (*char*) is stored internally as an integer, it can be cast to an int. It can also be cast to a *long*, byte, and *short* which are just different sized integers. An *int* can be converted to a *long* because it is going from a smaller data type to a larger one. A *long* can also be cast to an integer value but there are some limitations. If the value of the *long* is too large to fit into the range of values of an *int*, some of the information of the *long* will be dropped to make it fit, and the value that existed in the *long* will not be the same value that exists in the *int*. The same concept applies when converting a *short* to an *int* (a smaller datatype to a larger retains all information) and vice versa (a larger datatype to a smaller may loose information if the value is too large). This logic also applies when converting a byte to a *short* (smaller to larger) and vice versa (larger to smaller). A *char* and a *short* have the same storage size and can therefore be converted back and forth without any information loss. The size of *char* and *short* datatypes are equivalent to that of two bytes. Below is an example of casting.

```
public class CastTest {
    public static void main(String[] args) {
            int i = 2147483647;
            String s = “Hello”;
            char c = ‘c’;
            long l = 2147483648l;
            short sh = 0;
            byte b = 127;
            boolean bool = false;
            System.out.println((long)i); //from int to long
            System.out.println((int)l); //from long to int
            System.out.println((int)c); //from char to int
            System.out.println((short)c); //from char to short
            System.out.println((long)c); //from char to long
            System.out.println((byte)c); //from char to byte
    }

}
```

Prog. 4.10.1

Compile and run the program. Feel free to add more examples and test the program. Strings and *boolean*s can’t be cast to anything using static casting, but there are other built-in ways to convert from one type to another that aren’t discussed here (dynamic casting).

Chapter 5: Introduction to Scanner Class and Built In Methods

Section 5.1: Reading Input

We've talked about printing, datatypes, comments, and variables. If that was all there was to programming then it would quickly become boring. I'm going to introduce you to getting input from users and storing it and then we're going to get to the fun stuff.

The topic of classes will be coming later but for now all you need to know is that in order to get input from the user, the Scanner class must be imported. That can be done with the following statement

import java.util.Scanner;

All imports should occur before your class declaration. Now create a new program called ReadInput in a folder called 5.1.1. It should look like the following code.

```
import java.util.Scanner;
public class ReadInput {
   public static void main(String[] args) {


   }
}
```

Prog. 5.1.1

Now you have to declare a Scanner variable. It can be done like this

Scanner s;

Then the Scanner variable needs to be initialized like this

s = new Scanner(System.in);

This can be done in one step, so add the following code to you main block

Scanner s = new Scanner(System.in);

All you need to understand for now is that you're creating a scanner variable called *s* that reads input from *System.in* (the console). Then add the following code.

```
System.out.println("What is your name?");
String name = s.nextLine();
```

Nextline is a method. You've seen a java method before when using the *println* method to print to the screen. A method performs a set of actions in order to accomplish a specific goal. A Scanner has several read methods. The *nextline* method reads the next line of input until a newline character is encountered. The *next* method reads the next string of input until a space is encountered. The *nextInt*, *nextDouble*, *nextFloat*, *nextLong*, and *nextShort* methods read the next integer, *double*, *float*, *long*, or *short* value respectively. The *hasNext* method determines if there is more input to be read and returns true if there is and false otherwise. So add the final line of code and compile and run your program.

```
System.out.println("Hello "+name);
```

I'm sure you can already predict what will happen. Notice that the program doesn't do anything after printing the question. It waits for you to enter something. As *long* as you haven't entered anything, nothing will happen. Now here's a fun but simple program to demonstrate to you some of the scanner next methods. Create compile and run it.

```
import java.util.Scanner;
public class calc {
   public static void main(String[] args) {
            Scanner input = new Scanner(System.in);
                System.out.print("Enter the first value: ");
            int x = input.nextInt();
                System.out.print("Enter the second value: ");
            int y = input.nextInt();
            System.out.println(""+x+"+"+y+"="+ (x+y));
        }
}
```

Prog 5.1.2

Notice that the program uses *System.out.print()* instead of *System.out.println()* to request a value. This means that a newline character won't be printed. Since the newline character isn't printed, the cursor will remain on the same line until the user enters a value. Test the program out with different values to demonstrate that it works. Change the program so that it reads *double*s instead of integers. Have fun with it.

Practice Problems:

1. Write a Body Mass Index (BMI) calculator. Body Mass Index can be calculated by dividing weight in kilograms by height in meters squared. Weight in kilograms can be calculated from pounds by multiplying by 0.453592. Height in meters can be calculated from inches by multiplying by 0.0254.

   You'll find the solution to this problem on the website (www.jtjtackson.org).

Chapter 6: If Statement’s and Conditional Statements

Section 6.1: If Statements

Now it's time to have some fun. An if-statement is used to determine if a specified condition is true or not and performs some action if it is. Here's an example program

```
import java.util.Scanner;
public class Ifs {
    public static void main(String[] args) {
            Scanner in = new Scanner(System.in);
            System.out.println("How old are you?");
            int age = in.nextInt();
            if (age>=18) {
                    System.out.println("You're an adult");
            }
    }
}
```

Prog. 6.1.1

Write, compile, and run this program in a folder called 6.1.1. So what does this program actually do? It's pretty simple. A Scanner variable called *in* is created. Then the program waits for the user to enter an integer. If you enter something other than an integer, nothing will happen, until you enter an integer. You've seen all of this before. Once the integer value is entered on the console, the value is stored in the age variable. Here's where something new happens. The age variable is compared to the value 18 within the if-statement. If the value is greater than or equal to 18, "You're an adult" will be printed. Otherwise, nothing will happen and the program will continue after the if-statement. Since there is no code following the if-statement, the program will end. All if-statements have a condition, which is defined in the signature. The condition is an expression that must evaluate to a *boolean* result (true or false). If the condition evaluates to true, the line or lines of code within the if-statement will be performed. Otherwise they will be skipped and the next line of code after the statement will be performed. Although an if-statement must have a condition, it does not have to have the two curly braces. These are only required if the if-statement has more than one statement following the condition. If the curly braces are omitted then any statements after the first statement will not be considered part of the if-statement and will be run unconditionally. Either of the following would have been a valid replacement for the if statement above

```
if (age>=18)
    System.out.println("You're an adult");
```

Or

```
if (age>=18) System.out.println("You're an adult");
```

Good programming practice dictates that you always use the curly braces in case you later decide to add more lines of code. It's easier to avoid errors this way. If-statements can also have an optional block of code to perform in the event that the condition isn't true. Copy your *Ifs* program into a folder called *6.1.2* and replace the if-statement with the following code. The compile and run the new program.

```
if (age>=18) {
        System.out.println("You're an adult");
}
else {
   System.out.println("You're a child");
}
```

Prog. 6.1.2

An if-statement doesn't only have to have one or two possible branches. It can in theory have infinitely many branches. Copy the program into a folder called *6.1.3* and replace the if-statement with the following code.

```
if (age>=18) {
        System.out.println("You're an adult");
}
else if (age>=13) {
        System.out.println("You're a teenager");
}
else if (age>=10) {
        System.out.println("You're an adolescent");
}
else if (age>=4) {
   System.out.println("You're a child");
}
else if (age>=2) {
        System.out.println("You're a todler");
}
else {
        System.out.println("You're an infant");
}
```

Prog. 6.1.3

You can remove either the else block, one or more of the else-if blocks, or both and still have a valid program. So re-compile the program and then test it with various age values. Just like an if-block, an else-if block must have a condition. An else-if block will only be tested if the condition of the if-block and any previous else-if blocks have evaluated to false. An else-block doesn't have a condition. The code in an else-block is only performed if the condition of the if-block and any proceeding else-if blocks have evaluated to false. While else-if blocks can be added to an if-statement indefinitely, there comes a certain point when it becomes bad code. You'll have to use good judgment to determine when this point has been reached but a good rule of thumb is if the code has become too complicated for you, the writer, to follow, it's time to look for a way to simplify it.

An if-statement can be infinitely nested. This means that you can have an if-statement inside of an if-statement, inside of another one and so on, but the general rule of thumb is that any more than three levels are too many. Below is an example of a partial program with nested if-statements.

```
if (i==7) {
    if (j==9) {
            System.out.println("7 ate 9");
    }
}
```

Section 6.2: Conditional Statements

Conditional statements are similar to if-statements in what they seek to accomplish and they can be used to replace a simple if-statement for the sake of smaller code and conciseness. If there are more than two possible branches, it is better to use an if-statement. Here's a good example. A year is a leap year if it is evenly divisible by 400 or if it is evenly divisible by 4 and not evenly divisible by 100. A value is evenly divisible by another value if the value has a remainder of zero when divided by the other value. Below is a partial program example, using a conditional statement. Type it up as a full program, with whatever name you like, in a folder called *6.2.1* then compile and run it.

```
int year = 2016;

boolean is_leap_year;

is_leap_year = (((year%400==0) || (year%4==0 && year%100!=0)) ? true : false);

System.out.println("Is "+year+ " a leap year? "+is_leap_year);
```

Prog. 6.2.1

This looks a lot more complicated than it actually is. Let's start with an explanation of a basic conditional statement in java. A conditional statement is of the form

```
Condition ? value-to-return-if-true : value-to-return-if-false
```

In this particular case the condition is the following

```
((year%400==0 || (year%4==0 && year%100!=0))
```

This expression will return true if the year is divisible by 400 or if the year is divisible by 4 and not divisible by 100. You've seen this type of expression before. If the condition evaluates to true, true will be stored in the variable. If the condition isn't true, the value false will be stored in the variable. A conditional expression could return anything but the result when true and the result when false should have the same data type and if the value is being stored in a variable then the result type of the expression should match that of the variable. Otherwise an error will occur. This could have also been done.

```
int year = 2016;

String leap_year= (((year%400==0 || (year%4==0 && year%100!=0)) ? "is a leap year" : "is not a leap year");

System.out.println("The year "+year+ ""+leap_year);
```

Play around with this example. Instead of hard coding the year value, get the value from the user. This way the program can be tested with a variety of values.

Practice Problems

1. Write a program that gets an integer from the user and prints “The value x is even” if the value is even and “The value x is odd if the value is odd”. Hint: An even integer has a remainder of 0 when divided by two.
2. Write a program that takes an integer from the user that is between 0 and 100 and prints the grade.
    1. A (90-100)
    2. B (80-89)
    3. C (70-79)
    4. D (60-69)
    5. F (<60)

    Print a message if the value is not between 0 and 100
3. Write a program that prints the following under the specified conditions
    1. Print You’re Old enough to retire if the age is >= 62
    2. Print You’re Old enough to drink if the age is >=21
    3. Print You’re Old enough to marry and go to war if age >=18
    4. Print You’re Old enough for school if the age>=5
    5. Print You’re Old enough for preschool if age>=3
    6. Print You’re Not old enough for anything otherwise

Solutions to these problems can be found on the website (www.jtjackson.org)

Chapter 7: While Loops, For Loops, and Arrays

What if you had a task that you wanted to accomplish a certain amount of times or until a condition is no longer met? This is where loops come in. Say you wanted to print the first 50 integers. Sitting there writing 50 print statements would be extremely tedious. Instead you could write one inside a loop. Here’s an example.

Section 7.1: While Loops

Write and compile the following program in a folder called *7.1.1*.

```
public class loops {
   public static void main(String[] args) {
            int i = 1;
            while (i<=50) {
                  System.out.println(i);
                  i +=1;
            }
   }
}
```

Prog. 7.1.1

There are two new things going on in this program. Most importantly is the while loop. Similar to an if-statement, a while has a condition that it has to check. The difference between a while loop and an if-statement is that an if-statement checks the condition and performs the task if the condition is true and then moves on. A while-loop checks that a condition is met, and if this is the case, it repeatedly goes through the process of executing the code and re-testing the condition until the condition is no longer met. Once the while loop tests the condition and it is false, the program will jump to the line of code following the loop. If the condition is false the first time it is checked, none of the code in the loop is executed. The second new thing going on is the statement *i++*. This is simply a *short* hand way of saying *i = i + 1*. Another way of saying that would be *i+=1*. The difference is, *i++* always means *i = i+1* and *i+=* could be *i* equals *i* plus anything depending on the integer following it. Pay attention to the fact that the value *i* has increased. If this line of code were removed, you would have what is called an infinite loop. This is a loop that runs on forever. Eventually it would probably crash causing some kind of error. There are a few types of programs, such as a server, that require an infinite loop but we won't be writing any of those. If one of your programs has an infinite loop and you didn't specifically plan it, there is a problem. Play around with this value. Try printing all of the integers less than or equal to 50, counting by 5's. Try printing the first 50 integers counting by 5's. Pay attention to the wording because there is a difference. See what else you can do with the program.

While loops can be infinitely nested, meaning one inside of another but you should never use more than two or three (rarely) nested loops. Here's a partial program to demonstrate. You should write, compile, and run the full program (with the class, and main block plus the following code in the main block) in a folder called *7.1.2*. Pretend that there is a matrix with 3 rows and four columns. We seek to print the possible indexes.

```
int row = 3;
```

```
int col = 4;
int x = 1;
int y;
while (x<=row) {
   y = 1;
   while (y<=col) {
            System.out.println("(" +x+ "," + y+ ")");
            y++;
   }
   x++;
}
```

Prog. 7.1.2

So it's pretty simple. The outer loop will run as *long* as *x* is less than or equal to three. For each run through the outer loop, the *y* value is set to 1. Then, while *y* is less than or equal to four, *x* and *y* as coordinates are printed out and *y* is increased within the inner loop. Once y is greater than four, the inner loop is exited. Then the x value is increased, and the outer loops jumps back to the top to check the condition. This continues until the condition of the outer loop is no longer met.

Section 7.2: For Loops

For-loops aren't truly different from while loops. The difference is mostly in the syntax (the way it's written). Copy the *loops.java* program into a folder called *7.2.1* and replace the while loop with the following code. Also remove the *int i = 1;* line.

```
for (int i=1; i<=50; i+=1) {
   System.out.println(i);
}
```

Ex. 7.2.1

Notice the three sections within the parentheses of the for-loop. The first part (*int i=0*) is the variable declaration/initialization part of the loop signature. This part is optional but of course if you don't declare a variable here and you want to use it as part of the condition it must be declared above. If you declare a variable above, you can still assign a value to it in this section. If the variable declaration/initialization is skipped, a semi-colon must still be used to denote the end of that section of the signature. The second part is the condition. It is the same as the condition in the while loop. This part is required. The third part of the signature is the update portion of the for-loop. This is where the value or values used in the condition statement are updated. This part is optional but if it is skipped, a semi-colon (;) must still follow the conditional expression.

More than one variable can be declared and/or assigned a value in the declaration/initialization portion of the signature but they must all have the same type. Here's an example.

```
for (int i=0,j=1; i<20 && j<30; i+=2,j++) {
      //more code
}
```

Ex. 7.2.2

Notice that the type is specified once and then the variable are listed and given a value. Declaring and/or initializing variables separated by commas on one lines is also possible out side of a for-loop signature. It is the only way to declare and/or initialize multiple variables within the signature of the for-loop. Nothing new is going on in the condition section. It is optional to make both variables a part of the condition. In fact your condition statement main not relate to the variables you initialized at all. It just depends on what you're trying to accomplish. Play around with this loop the way you did with the while loop, trying to accomplish the same tasks.

Section 7.3: Single Dimensional Arrays

What if you needed to store 10 variables of the same type (maybe 10 names as strings or 10 ages as integers)? While creating 10 variables is a bit tedious, it may not seem like too big a task. So what happens if you needed to create 100 or 500 variables of the same type? Now creating 500 variables would be extremely tedious, time consuming, and hard to manage. But it doesn't have to be. This is where arrays come in. An array can be thought of as a series of boxes, in order, numbered zero through *n-1*. Each box can contain a value. Once you choose your number of boxes, you can't add any or take any away but you can change the contents. Let's say you had 5 names and 5 ages you needed to store. Here's how you'd do it. You would do something like this

String[] names = new String[5];

int[] ages = new int[5];

Ex. 7.3.1

For simplification I combined the declaration and initialization steps but it could have been separate, just like with any other variables. By saying *String[] names;* and *int[] ages;* I am declaring a String array called names and an integer array called ages. Until the variables are initialized they don't have any size. Once they are initialized, their sizes can't be changed. The names and ages arrays now each has five "boxes" where you can place values but so far you haven't placed any values in these boxes. An array of length 5 has 5 indexes. Its first index is zero and its last index is *length-1* (four in this case). To change the value of one of the boxes (locations) in an array, you have to use the location's index. Here's an example

names[0] = "George";

If you try to access a location in an array with an invalid index, an error will occur. Like regular variables, you can't use an array before you initialize it.

You've seen how to initialize array variables using the new keyword and specifying the size but there is another way to do it. Create a java file with any name you wish, in a folder called *7.3.1* and add the following lines of code to the main block.

String[] names = {"James","Robert","Stanley","Shirley","Janice"};

Int[] ages = {5,10,13,20,28};

Prog. 7.3.1a

By initializing the variables this way, you are indirectly setting their sizes, and giving each of the available slots a value. The values can be changed. You already know that the size cannot. Here's an example of accessing the values in the names array, after initializing them above, to print James's name and age.

System.out.println(names[0]+ " is "+ages[0]+ " years old");

To print Janice's name and age you would do this

System.out.println(names[4]+ " is "+ages[4]+ " years old");

What if you wanted to print all of the names and there corresponding ages? Add the following lines to your program to print the names and ages.

```
for (int i = 0; i<names.length; i++) {
        System.out.println(names[i]+ " is "+ages[i]+ " years old");
}
```

Prog. 7.3.1b

Note that this could have just as easily been accomplished with a while-loop but a for-loop somewhat simplified the code. The for-loop declares the variable *i*, initializes it to 0 and prints out the name and age at index *i*, as *long* as *i* is less than the length of names. Note that using *arrayname.length* returns the length of an array. This loop operates under the assumption that any index that is valid in *names* is also valid in *ages*. If you were writing a loop that indexes two arrays, where this assumption might not be valid, then your condition would have to look something like this

```
i<names.length && i<ages.length
```

If the first condition in an and-expression evaluates to false, the second condition won't even be checked because the result must be false. So putting the comparison of the variable with the array whose length is shorter will save a miniscule amount of time. The time savings may become more significant in a larger array. As a side note, if the first condition of an or-expression evaluates to true, the second condition isn't checked because the result must be true.

What if you wanted to change a value in the *ages* array? It's easy. Let's say today is James' birthday and he has just turned six years old. So what would you do? You could simply do this

```
ages[0] +=1;
```

or

```
ages[0]=6;
```

As you can see, assigning a value to an index in an array would follow the same rules that apply to assigning a value to another variable, of the same type.

Practice Problems

1. Print the first 20 even numbers (Hint: count by two's from 0)
2. Print the first 20 odd numbers (Hint: count by two's from 1)
3. Take an integer from the user and print it's timetables from 0 to 12
4. Print the first 100 integers backwards (100, 99, 98, …). (Hint: The update condition doesn't have to always be an increase in the integer (when an integer is used). It can also be a decrease. Try i=i-1 or i—)

Solutions to these problems can be found on the website (www.jtjackson.org).

Chapter 8: Strings

Section 8.1: Introduction to the String Class

All of the other built in data types (*int*, *double*, *etc.*) in java are what would be called primitive but Strings are actually a class type. This means that they come with their own methods and properties. You already used a class type when you created a Scanner. Creating a new *String*, *Scanner*, or other class type is creating an instance of that type. An instance of a class is called an object.

You've only seen one way to initialize a string, but there is another way to do it. You're used to using the statement ***String s = "mystring";*** to create *String*. What you didn't know about this method of creating a new string is that you often aren't actually creating a new string. Java does something called string interning when using the above method to create *String*s. String interning means that only one copy of each distinct string value is created and these copies are immutable. These immutable *String*s are stored in the string pool. A new string is only created if the string doesn't already exist in the string pool. Otherwise, all strings assigned the same string value, will reference the same string in the string pool. The second method of creating a string is as follows.

String s = new String("mystring");

You'll recognize a similar syntax from when you create Scanner objects. That's because you are creating a new String object. This object will have its own reference whether the string exists in the string pool or not. The code following the new keyword looks like a method but it is actually a constructor. Constructors are used to create new objects.

Section 8.2: Length Method

One important String method that you'll find yourself using over and over again is the string *length* method. Here's an example program. Type it up in a folder called *8.2.1*, compile it, and run it.

```
import java.util.Scanner;
public class strs {
   public static void main(String[] args) {
            Scanner input = new Scanner(System.in);
               System.out.println("Enter a string to determine it's length");
            String str = input.nextLine();
            System.out.println("The string "+str+" has a length of "+str.length());
   }
}
```

Prog. 8.2.1

Now you can play around with this, entering various strings to see that it works.

So far you've only seen methods that don't have any arguments. So what is an argument? It is a value that is passed to a method. A method takes this value and performs some action using it. A method can have zero or more arguments. If more than one argument is passed to a method the arguments will be separated by commas. An argument can be any type of variable or value as *long* as it's the correct type for that particular method. A method accepts its arguments in a specified order. For example, if a method accepted an integer as it's first argument and a string as it's second argument, then they must be passed in that order. The method discussed in the next section will accept one argument.

Section 8.3: CharAt Method

In order to get a character at a particular index in a string, you have to know the index. A String's first index is zero and its last index is *length-1* (like an array). The *charAt* method can be used to get the character at a specified index in a String. Copy your *strs* program from the last section into a folder called *8.3.1* and replace that code with the following code.

```
import java.util.Scanner;
public class strs {
   public static void main(String[] args) {
            Scanner input = new Scanner(System.in);
               System.out.println("Enter a string");
            String str = input.nextLine();
            int max = str.length()-1;
            System.out.println("Enter an integer between 0 and "+max);
            int index = input.nextInt();
            if (index<0 || index>max) {
                     System.out.println("You have entered an invalid index");
            }
            else {
                     System.out.println("The character at index "+index+" is
"+str.charAt(index));
            }
   }
}
```

Prog. 8.3.1

As you can see, there's a decent amount going on in this program, but you have seen most of this before. A new Scanner is created. Then the user is prompted to enter a string. Once a string is entered, the user is prompted to enter an integer. Next the program checks if the index entered is within the valid range. If it is not, the user is notified. Otherwise, the character at the specified index is printed. As you can see the *charAt* method takes one argument, an integer. In this case a variable was used to pass the index to the method but the programmer could have just as easily passed a value directly like this

```
str.charAt(0)
```

It is often better practice to store important values in variables because if the value is used in multiple places, it can be tedious and error prone to go through and replace each use of the value. It is easier to change the variable in one place and have that change reflected

throughout the entire program.

Many of the programs in this book involve reading input from the user instead of hardcoding the values into the programs because it makes the program more dynamic. If the values were hardcoded into the code then you would have to repeatedly change the program to test different values. I wanted you to be able to compile the program once and test it many times. Beware of the danger of taking user input though because they can enter whatever they want. You have to be sure that the values are valid for your particular program, or your program could crash. We insured the validity of user data in the above program. If the *charAt* method had been passed an invalid index value, an error would have occurred, stopping the program from continuing. This is why we checked the validity of the value before passing it to the *charAt* method.

Section 8.4: Equals Method

You may or may not have noticed that up until now, we have never compared two Strings, using the == operator. Let's try that now. Copy your *strs* program into a folder called *8.4.1* and change the code to reflect the following code.

```
public class strs {
   public static void main(String[] args) {
            String str1 = "string";
            String str2 = new String("string");
            boolean equal = (str1==str2);
            if (equal) {
                  System.out.println("The strings "+str1+ "and"+str2+ "are
equal");
            }
            else {
                  System.out.println("The strings "+str1+ " and"+str2+ " are
not equal");
            }
   }
}
```

Prog. 8.4.1

Compile and run the program. The *not equal* string should be printed. Now you may go back and examine the strings and determine that they are in fact equal. So what is the problem? As I mentioned to you before, a *String* is an object not a primitive value. It is more complicated than a simple integer, character, or *boolean* and therefore cannot be compared using the == operator. When used on an object, the == operator determines whether two references are equal, meaning whether or not two variables point to the same location in the memory space (and therefore the same object). If you were to create two strings using the following code and compare them using the == operator, they would be considered equal because they would point to the same reference in the string pool.

```
String str = "mystring";
String str2 = "mystring";
```

The string class has a method that determines if two strings are actually equal, regardless of reference, called the *equals* method. Here's an example. Copy the last *strs* program to a folder called *8.4.2* and make the change (it's a small one).

```
public class strs {
```

```
    public static void main(String[] args) {
            String str1 = "string";
            String str2 = new String("string");
            //This is the only difference. No need to copy this comment
            boolean equal = str1.equals(str2);
            if (equal) {
                    System.out.println("The strings "+str1+ "and"+str2+ "are
equal");
            }
            else {
                    System.out.println("The strings "+str1+ "and"+str2+ "are
not equal");
            }
    }
}
```

Prog. 8.4.2

Compile and run the program. We've only changed one line of code, but as you should see, it makes all the difference. The *equals* method is used on one of the strings and is passed the other string as an argument. For this particular method it doesn't matter which string is which but this is not always the case.

Section 8.5: CompareTo Method

You've seen that the String *equals* method compares two strings and returns a *boolean* value indicating whether they are equal or not. Sometimes you may need more information than this. That is where the *compareTo* method comes in. The *compareTo* method returns an integer. If two strings are equal, it returns a zero. If the string being compared would come after the string being passed in alphabetical order (it's lexicographically greater), then an integer greater than zero would be returned from the method. If the string to be compared would come first in alphabetical order (it is lexicographically smaller) then a value less than zero will be returned from the method. Create a new *strs* program in a folder called *8.5.1* and write the following code, compile, and run it.

```
public class strs {
   public static void main(String[] args) {
            String str1 = "apple";
            String str2 = "apple";
            String str3 = "bike";
            System.out.println("str1 compared to str2: "+str1.compareTo(str2));
            System.out.println("str2 compared to str3: "+str2.compareTo(str3));
            System.out.println("str3 compared to str2: "+str3.compareTo(str2));
   }
}
```

Prog. 8.5.1

Feel free to change the strings, or change the program so that the user can enter the strings.

There are more string methods but this was just a brief of methods you'll repeatedly use.

Practice Problems

1. Read a word from the user and print it backwards (Hint: You'll need the String methods *charAt* and *length* as well as a for-loop or while-loop);
2. Write a program that reads two strings from the console and writes a message that indicates if the two strings are equal or which is greater (Hint: You'll need a scanner and an if-statement)

   The solutions to these problems can be found on the website (www.jtjackson.org).

Chapter 9: Math Class

Java has a Math class that can be use to perform a variety of math calculations. Below is a list of various methods of this class, their arguments, and their return values

Method	Argument	Returns
abs()	This method accepts a single argument of type *double*, *float*, *int*, or *long*. This is actually not one method but several, one for each of the mentioned data types (same names different arguments (overloaded methods chpater 11))	This method returns the absolute value of the argument. The return type is the same as the argument type for each method
acos	Accepts a *double* as it's argument	Returns a *double* value between 0.0 and pi that is the arc cosign of the argument
asin	Accepts a *double* as it's argument	Returns the arc sine of a value in the range -pi/2 – pi/2 as a *double*
atan	Accepts a *double* as it's argument	Returns the arc tangent of a value within the range of -pi/2 – pi/2 as a *double*
atan2	Accepts two *double* values as arguments (y,x)	Returns the angle theta from the conversion from rectangular coordinates to polar coordinates as a *double*
cos	Accepts a *double* as an argument	Returns the cosine of the argument as a *double*
floor	Accepts a *double* as an argument	Returns the largest *double* value that is less than or equal to the argument and equal to a mathematical integer (rounds down)

log	Accepts a *double* as argument	Returns the natural logarithm of the argument as a *double*
log10	Accepts a *double* as argument	Returns the base 10 logarithm of the argument as a *double*
max	Accepts either two *doubles*, two *ints*, two *longs*, or two *floats* as args (not one method, actually four overloaded methods)	Returns the larger of the two given values. Return type is that of the two passed in values
min	Takes either two *doubles*, two *ints*, two *longs*, or two *floats* as args (not one method, four overloaded methods)	Returns the smaller of the two given values. Return type is that of the passed in values.
random	Doesn't take any arguments	Returns a random *double* value greater than or equal to 0.0 and less than 1.0 as a *double*
round	Accepts a single argument either a *double* or a *float* (not one method but two overloaded methods)	Returns the closest *long* (when arg is a *double*) or the closest *int* (when arg is a *float*) to the argument
sin	Accepts a *double* as argument	Returns the sine of the argument as a *double*
pow	Accepts two *doubles* as arguments	Returns the value of the first argument raised to the power of the second argument as a *double*
tan	Accepts a *double* as argument	Returns the tangent of the argument as a *double*

Figure 9.1

The math class has quite a few more useful methods but I chose to list the ones that I believe would be more useful for you. Below is an example of the use of the math class

```
public class MoreMath {
    public static void main(String[] args) {
        double abs_double = Math.abs(-4.0);
        float abs_float = Math.abs(-5.1f);
        long abs_long = Math.abs(15l);
        int abs_int = Math.abs(13);
        System.out.println("abs_double: "+ abs_double+ " abs_float: "+
    abs_float);
        System.out.println("abs_int: "+abs_int+ " abs_long: "+ abs_long);
        double max_double = Math.max(17.2,35.0);
        float max_float = Math.max(13.7f,13.6f);
        long max_long = Math.max(25l,47l);
        int max_int = Math.max(63,62);
        System.out.println("max_double: "+ max_double+ "max_float: "+
    max_float);
        System.out.println("max_int: "+ max_int + "max_long: "+ max_long);
        long round_long = Math.round(100.07); //accepts a double returns a
    long
        int round_int = Math.round(95.13f); //accepts a float returns an int
        System.out.println("round_long: "+round_long+ "round_int: "+
    round_int);
            double min_double = Math.min(17.2,35.0);
        float min_float = Math.min(13.7f,13.6f);
        long min_long = Math.min(25l,47l);
        int min_int = Math.min(63,62);
        System.out.println("min_double: "+ min_double+ "min_float: "+
    min_float);
        System.out.println("min_int: "+ min_int+ " min_long: "+ min_long);
    }
}
```

Prog. 9.1

Write the program in a folder called *9.1* and compile and run it.

Chapter 10: Files and I/O

Section 10.1: Input and Output (I/O)

A stream is a sequence of data coming from some source. There are two types of streams. There is an input stream which is used to read data from the source and an output stream which is used to write data to a source. Java has byte streams which perform input and output on 8-bit bytes. The classes most commonly used with byte streams are *FileInputStream* and *FileOutputStream*. Java has character streams which perform input and output on 16 bit Unicode characters and are most commonly used with the *FileReader* and *FileWriter* classes. Then there are the standard streams. Standard input is used to feed data to a program. A keyboard is usually used as the standard input stream and represented as *System.in*. You have seen *System.in* before, when creating a scanner object that reads data from the console. Standard Output is used to output the data of programs. The computer screen is usually used as the standard output stream and represented as *System.out*. You've seen *System.out* on many occasions when writing a print statement. Now you know enough to understand that *System.out* is the object that an action is performed on and *println* is the method (or effectively, the action performed). You'll also realize that the *println* method can accept a *String*, *double*, *char*, *long*, *short*, or *byte* as an argument. Standard Error is used to output the error data produced by programs. The computer screen is usually used as the standard error stream and represented by *System.err*. The java *FileInputStream* is used to read data from files and the *FileOutputStream* is used to create a file and write data to it. There are way too many I/O classes in Java to cover in any detail in this Quick Guide to Java Programming. We will instead focus on the File class.

Section 10.2: The File Class

Up until this point, we've been reading data from the command line but this isn't always the best route to take. Sometimes you may require your program to use predefined data, possibly written by another program in which case you would use a file.

In order to use the File class, you must first import it as follows.

import java.io.File;

To create a File object, you would follow the same steps you use to create any other object. When creating a new File, the constructor takes a string containing the name of the file, as an argument. A file can be an actual file or a directory/folder (basically a File containing a list of other files). Therefore, the name can be the path to a file or the path to a directory. This path can be an absolute path or a relative path.

An absolute path is a path that contains all of the information of it's location from the starting point to the ending point. An absolute path in a Unix System is a path starting with a forward slash (/), meaning that it starts at the root directory. An absolute path in a Windows System is a path starting at some drive (say *C:*).

A relative path is a path that is assumed to start from the current directory and therefore, only includes the path information after this point. If the File is not a directory, then the path passed to the constructor will end in a file name (and could optionally only contain a file name). For example, *C:\user\temp\filename.txt (Windows)* or */usr/tmp/filename.txt (UNIX)*. Below is an example of using the file class

```
import java.io.File;
public class UseFiles {
   public static void main(String[] args) {
        /*Note that using the file name alone is a relative path and the path is assmed to start from the current directory*/
        File f = new File("/tmp/user/java/bin");
        boolean success = f.mkdirs();
        String stat = (success? "has" : "has not");
        System.out.println("The directory "+f.getAbsolutePath()+stat+ " been successfully created");
        File file = new File("/tmp/user/java");
        if (file.isDirectory()) {
            String[] paths = file.list();
            File[] files = new File[paths.length];
            for (int i=0; i<files.length; i++) {
                files[i] = new File(paths[i]);
```

```
                    }
                    for (int i=0; i<files.length; i++) {
                        System.out.println("File "+(i+1)+ ": 
"+files[i].getPath());
                    }
                }
            }
        }
```

Prog. 10.2.1

You'll notice the creation of a File object as well as some new methods in this example program. Also notice that while the path used would be an absolute path in UNIX (Starting from the root directory and ending at the target directory), the path would be a relative path on a Windows system (not starting from a particular drive i.e. *C:*). The *mkdirs* method creates the target directory (bin) and creates all of its parent directories (*tmp*, *user*, and *java*). It returns true on success and false otherwise. The *mkdir* method could have also been used, which only creates the target directory (bin) but this wouldn't have worked if the parent directories didn't already exist. The *getAbsolutePath* method is self explanatory. It returns a string representing the absolute path of the File. The *isDirectory* method returns true if a File is a directory and false otherwise. There is also a corresponding *isFile* method that returns true if a file is a regular File (and therefore not a directory) and false otherwise. The *list* method returns an array of strings of the paths of files and/or directories in a directory. I retrieved this list of files and then used these path strings to create an array of files. I then proceeded to print the paths of these files using the *getPath* method, which basically returns the relative path of a file or directory. This is a bit repetitive seeing that I could have simply printed the list, but then I wouldn't have been able to demonstrate the *getPath* method. There is also a *listFiles* method that would have directly created an array of File objects but once again, that would have taken away some of the demonstration. The method *exists* returns true if a file exists and false otherwise. The *createNewFile* method creates a File if it doesn't *exist*. There are many more methods for the File class. A simple google search of the java File class will return a more comprehensive list (check out tutorialspoint.com), but I just wanted to familiarize you with a few of the most common ones. Now let's turn our eye to file reading and writing.

Section 10.3: Writing to Files

In order to write to a file, the *PrintWriter* class can be used. Note the importance of closing a *PrintWriter* object, after you have finished using it. If you fail to do so, your program may not crash, but it also may not behave as you expect. Below is an example of using the *PrintWriter* class. Write the program in a folder called *10.3.1*, compile, and run it.

```
import java.io.PrintWriter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
public class WriteFiles {
    public static void main(String[] args) {
            File file = new File("temp.txt");
            try {
                    file.createNewFile(); //create it
            }
            catch (IOException ioe) {
                    System.out.println(ioe.getMessage());
                    System.exit();
            }
            PrintWriter p = null;
              try {
                    p = new PrintWriter(file);
                    p.println("Hello World");
          p.println("How's it going");
              }
              catch (FileNotFoundException fnf) {
          System.out.println(fnf.getMessage());
              }
            p.close();
    }
}
```

Prog. 10.3.1

There are a few new things going on. First, a file object is created. The program then attempts to create the file if it doesn't exist. We'll go further into try-catch blocks in chapter 12 but for now you just need to know that there are certain kinds of pre-defined errors that a program can encounter (such as Input Output Exceptions and File Not Found Exceptions) that you are required to check. Checking these errors means that the programmer specifically defines steps to take in the event that these errors occur. Notice that we had to import the two exception classes that we used, along with the *PrintWriter* and *File* classes. When attempting to create the file, if an *IOException* occurs I use the *getMessage* method on the *IOException* object *ioe* to get the exception message (as a string) and print it using the *println* method and then the *exit* method is used to end the program. You also haven't encountered null yet. For now just think of it as a place holder for object variables. Assuming the program is still running, we attempt to create a *PrintWriter* object using the file. We use a try-catch block to catch a potential file not found exception. Then we used the *println* method to add a few lines to the file and finally we close the *PrintWriter* object. The file should have been created in the same folder as your java program. Before we move on to the next section, I want to bring something to your attention. If you were to run this program a second time, it wouldn't append the lines to the file again. It would instead overwrite the file. If you wanted to append to the file, you could use the *FileWriter* class instead of the *PrintWriter* class. The *FileWriter* class has a constructor that accepts two arguments, a file, and a *boolean* value that indicates whether the file should be appended to (happens when the *boolean* value is true), or overwritten (when value is false). Copy your previous program into a folder called *10.3.2* and make the necessary changes.

```
import java.io.File;
import java.io.Exception;
import java.io.FileWriter;
public class WriteFiles {
    public static void main(String[] args) {
            File file = new File("temp.txt");
            FileWriter f = null;
            try {
                    file.createNewFile();
                    f = new FileWriter(file,true);
                    f.write("Hello World\r\n");
                    f.write("How's it going\r\n");
                    f.close();
            }
```

```
            catch (IOException ioe) {
                System.out.println(ioe.getMessage());
            }
        }
    }
```

Prog. 10.3.2

Notice the \r and \n characters. Those aren't literal characters. The \r represents a carriage return and \n represents a linefeed. Depending on your system, one or both of these characters are used to start a newline. If you're getting some unusual results, google the proper sequence for your system. Now if you were to run this program multiple times, you should see those lines added multiple times.

Section 10.4: Reading a File

A file can be read just as you would read input from the console, using a Scanner. Copy one of your temp.txt files to a folder called *10.4.1*, write the following program in the folder, compile it, and run it. If your folder doesn't contain a temp.txt file, your program won't work.

```
import java.util.Scanner;
import java.io.File;
public class ReadFiles {
    public static void main(String[] args) {
            File file = new File("temp.txt");
            if (file.exists()) {
                    Scanner in = new Scanner(file);
                    String line = "";
                    for (int i=0; in.hasNext(); i++) {
                            line = in.nextLine();
                            System.out.println("Line "+(i+1)+ ": "+line);

                    }
                    in.close();
            }
    }
}
```

Prog. 10.4.1

As demonstrated above, your use of the Scanner class to read from a file is no different that reading from the console. I don't believe that you've seen the *hasNext* method yet. It returns true if there is more content to be read, following the content (if any) that has already been read. I think I've neglected to mention to you so far that a scanner object can also be closed once you're finished with it, using its own *close* method. It is a good practice to adhere to.

Chaptere 11: Classes

Section 11.1: Properties (Data Fields) of a Class

We've talked some about classes but we're going to go into a little more detail now. You may already have realized that any java program must have a class and all other information (other than the import statements) go inside the class. There is a lot more to classes than that. A class defines the properties (data fields) and behaviors (methods) of a particular type of object. For example, let's say you had a dog class. What properties would a dog have. It would have the properties *breed*, *color*, *name*, and *age* for starters. So here's an example of a dog class.

```
public class Dog {
    public String color;
    public String breed;
    public int age;
        public String name;
}
```

Prog. 11.1.1

One thing you'll notice is that the main block isn't here. I've been calling it a block up until now but in fact, it's a method. The main method is not actually required. When the java virtual machine runs a java program it searchers for a main method to run. You can compile the above class just as before but since it doesn't contain a main method, attempting to run it would cause an error. Something else that you may notice is that the variable declarations are now preceded by the keyword public. The three visibility modifiers available in java are *private*, *public*, and *protected*. They determine where a variable or method can be accessed. We'll get back to that later. For now just know that you can only use a visibility modifier when declaring a variable inside a class, but not inside a method (like the main method). This is why we hadn't been doing it before. A variable declared inside of a method can only be accessed in that method so a visibility modifier would not be necessary.

Section 11.2: Creating an Instance of a Class

Now let's get back to our dog class. Copy the class to a folder called *11.2.1* and then we're going to make the changes demonstrated in the following code, including adding a main method.

```
public class Dog {
    public String color;
    public String breed;
    public int age;
    public String name;
    public static void main(String[] args) {
            Dog dog = new Dog();
    }
}
```

Prog. 11.2.1

So what have we done here? Inside of the main method, we have created an instance of our dog class. Creating a class is equivalent to creating a new type. Once a class has been defined you can create as many instances of this class, or objects, as you want. Each object has its own copy of the properties/data fields. In this case the properties are *color*, *breed*, *name*, and *age*. Just like you can create a new string which will have its own length and other properties, and changing one string will not affect another, the same can be said of data types created by you. You'll see this in the next section, with our new changes to the code. Before we get to that, look at the above code one last time. Notice the *Dog()* portion of the code. That is our no-argument constructor which creates a new instance of our Dog class. Just like with other classes, the constructor must always have the same name as the class. We'll talk more about constructors soon.

Take another look at the data fields of a dog class. The fact that classes are made of various data fields and methods is the reason why they can't be compared using the == operator. This operator is for simple data types with only one value (ex. *char*, *int*, *boolean*, etc).

Section 11.3: Accessing the Properties of a Class

Copy your program above to a folder called *11.3.1* and make the necessary code changes to get the program described below.

```
public class Dog {
    public String color;
    public String breed;
    public int age;
    public String name;
    public static void main(String[] args) {
            Dog fido = new Dog();
            Dog fifi = new Dog();
            fido.name = “Fido”;
            fido.age = 3;
            fido.breed = “German Sheppard”;
            fido.color = “Gold”;
            fifi.name = “Fifi”;
            fifi.age = 2;
            fifi.color = “white”;
            fifi.breed = “poodle”;
            System.out.println(“” + fido.name + “ is a ”+ fido.age+ “ year old ” +fido.color+“, ”+ fido.breed);
            System.out.println(“” + fifi.name + “ is a ”+ fifi.age+ “ year old ” + fifi.color+“, ”+ fifi.breed);
    }
}
```

Prog. 11.3.1

Write, compile, and run this program. Notice that one way to set the properties of an object is to assign a value using the statement *object.propertyName = value;*. Notice that setting the properties of one object of the dog type doesn’t affect the other.

Section 11.4: No argument Constructor

Copy your last version of the Dog class to a folder called *11.4.1*. Then, make the following changes to the code, and let's discuss.

```
public class Dog {
    public String color;
    public String breed;
    public int age;
    public String name;
    public Dog() {
            color = "Brown";
            breed = "Mutt";
            age = 1;
            name = "ToTo";
    }
    public static void main(String[] args) {
            Dog fido = new Dog();
            Dog fifi = new Dog();
            fido.name = "Fido";
            fido.age = 3;
            fido.breed = "German Sheppard";
            fido.color = "Gold";
            fifi.name = "Fifi";
            fifi.age = 2;
            fifi.color = "white";
            fifi.breed = "poodle";
            System.out.println("" + fido.name + " is a "+ fido.age+ " year old " +fido.color+", "+ fido.breed);
            System.out.println("" + fifi.name + " is a "+ fifi.age+ " year old " + fifi.color+", "+ fifi.breed);
    }
}
```

Prog. 11.4.1

We have just added a constructor to the Dog class. A constructor is declared using an

optional visibility modifier followed by the name of the class, then a set of parentheses which can hold a list of zero or more parameters. The arguments passed in when calling a method or constructor are parameters when defining a method or constructor. Finally a set of curly braces follow to define the block of the constructor where its code lies. In this particular constructor, I gave all of the properties default values. Each Dog object that is created will have its properties set to these values until they are changed. Doing this in the constructor insures that a Dog object will always have values for its properties. This particular constructor is called a no argument constructor because it doesn't define a parameter list in its signature.

If you don't define a constructor in the code, the java virtual machine will use what is called the default constructor of the class. This is a built-in no-argument constructor that can be used to create an instance of a class without doing anything else, when no constructor is defined. I've mentioned this before but wanted to reiterate that when calling the constructor of an object, the constructor name is always preceded with the new keyword.

Section 11.5: A Constructor with Arguments

Copy the latest version of the program, to a folder called *11.5.1* and make the necessary changes to get the code below.

```
public class Dog {
    public String color;
    public String breed;
    public int age;
    public String name;
    public Dog(String name, String breed, String color, int age) {
            this.name = name;
                this.color = color;
            this.breed = breed;
            this.age = age;
    }
    public static void main(String[] args) {
            Dog fido = new Dog("Fido", "German Sheppard", "Gold",3);
            Dog fifi = new Dog("Fifi", "poodle", "white",2);
            System.out.println("" + fido.name + " is a "+ fido.age+ " year old ",
+fido.color+" "+ fido.breed);
            System.out.println("" + fifi.name + " is a "+ fifi.age+ " year old " +
fifi.color+", "+ fifi.breed);
    }
}
```

Prog. 11.5.1

Notice that the dog constructor now has parameters. A parameter list is a comma separated list of types and names. This particular constructor takes as arguments, values for each of the properties and uses them to set the properties of the Dog. When a parameter is defined in a constructor or method, you are effectively defining a local variable that is only accessible within that constructor or method. When a method or constructor with arguments is called, you are passing these arguments to those local variables for the constructor or method to access. Notice that the names of the parameters are the same names as the class' data fields. Now notice the keyword *this* that we are using. It is not required that parameter names be the same as data field names but when this is the case, or when a local variable defined within the constructor or method has the same name as a data field, then the this keyword helps java to distinguish between the two. Saying *dataFieldName* alone would be referencing the local variable because the java virtual

machine is always going to look locally for a variable before looking globally. But saying *this.dataFieldName* means that you are accessing the class variable of this particular object. Variables defined in the class instead of in a method or constructor, are global and can be accessed anywhere. On the other hand, variables defined within a method or constructor can only be accessed there. This is a concept called scope that we will get more into later. You can use the keyword *this* whether local variables with the same names exist or not, if you just want one section of code to follow the same patterns as another.

Section 11.6: Overloading a constructor

A class can have more than one constructor. They will of course have the same name and possibly the same visibility modifier but what set's them apart is their parameter list. If a class has more than one constructor, the constructors are said to be overloaded constructors. All constructors must differ in the type, order, and/or number of parameters. This is how the java virtual machine distinguishes between them. In the following example, I'll put the original, no argument constructor that we created before back and leave the new one as well, then use both in the main method. Copy one of your old Dog classes to a folder called *11.6.1* and make the necessary changes.

```
public class Dog {
    public String color;
    public String breed;
    public int age;
    public String name;
    public Dog() {
            color = "Brown";
            breed = "Mutt";
            age = 1;
            name = "ToTo";
    }
    public Dog(String name, String breed, String color, int age) {
            this.name = name;
                this.color = color;
            this.breed = breed;
            this.age = age;
    }
    public static void main(String[] args) {
            Dog fido = new Dog("Fido", "German Sheppard", "Gold",3);
            Dog fifi = new Dog();
            fifi.name = "Fifi";
            fifi.age = 2;
            fifi.color = "white";
            fifi.breed = "poodle";
```

```
            System.out.println("" + fido.name + " is a "+ fido.age+ " year old "
+fido.color+", "+ fido.breed);

System.out.println("" + fifi.name + " is a "+ fifi.age+ " year old " + fifi.color+",
"+ fifi.breed);

   }

}
```

Prog. 11.6.1

Now compile and run the program. We used the no argument constructor to create *fifi* and then manually set her properties, but used the other constructor to create *fido*.

Section 11.7: Programmer Created Methods

I mentioned before that along with properties (data fields) a class also has behaviors (methods). Not only can you use built-in methods but you can create your own. The same goes for classes. It's time to create a few of those methods. Copy your program a folder called *11.7.1* and make the necessary changes.

```
public class Dog {
    public String color;
    public String breed;
    public int age;
    public String name;
    public Dog() {
            color = "Brown";
            breed = "Mutt";
            age = 1;
            name = "ToTo";
    }
    public Dog(String name, String breed, String color, int age) {
            this.name = name;
                this.color = color;
            this.breed = breed;
            this.age = age;
    }
    public void speak() {
            System.out.println(""+name+ " says Bark!");
        }
    public static void main(String[] args) {
            Dog fido = new Dog("Fido", "German Sheppard", "Gold",3);
            Dog fifi = new Dog("Fifi", "poodle", "white",2);
            fido.speak();
            fifi.speak()
        }
}
```

Prog. 11.7.1

Compile and run the program. Notice the code that we have added. It looks something like a constructor but it isn't. It's a method. You'll notice a slight change, that is, the void keyword. This is the return type. All methods must have a return type. A constructor, on the other hand, doesn't require you to explicitly specify a return type because it will always return an instance of the class. A method can return any of the primitive return types, any class types, and arrays. As a side note, just like you can create an array of anything including a programmer defined type like *Dog*.

The return type void is used when a method doesn't return anything. As you can see, the method defined above just prints a simple statement. Notice that in the main method, the speak method is called just like any other method. You just use the following syntax

Object.method(optional-argument-list)

Methods are defined using an optional visibility modifier, followed by the return type, the name of the method, and the (possibly empty) parameter list. Then of course there are the curly braces that hold the method code within them.

Section 11.8: Static Variables and Methods

Let's talk a little about the main method. You'll notice a slight difference between my description of how a method is defined, and how the main method is defined, that is, the static keyword before the return type. That is because the main method is (surprise) a static method. A static method belongs to a class and not a specific instance of it. They don't affect any of the instance data fields and can't operate on a particular object. They usually perform some action on the data passed in and then return. The speak method that we've defined above is an instance method. As you've already seen, it would be called using the syntax *objectName.instanceMethod(optional-parameters)*. On the other hand, a static method would be called using the syntax *className.staticMethod(optional-parameters)*. Just like you can create a static method, you can also create a static data field (class variable). Each instance of a class will not have its own copy of a static data field. Instead they will all share one copy of the field.

You know that the java virtual machine calls the main method of a class. So can the main method take arguments like other methods? If so, how does it get these arguments? Go back to one of the programs that include a main method. Do you notice the *String[] args* in the signature of the main method? This means that the method accepts an array of strings as its argument. So where does the array of strings come from? They come from you. But you may argue that up until now, you haven't passed any arguments to the main methods of any programs. You would be right. When you don't specify a list of arguments, a String array of length zero is passed to the main method. So how would you pass arguments? Copy your first program, *HelloWorld*, to a folder called *11.8.1*, and make the changes necessary changes to get the program below.

```
public class HelloWorld {
   public static void main(String[] args) {
            if (args.length>0) {
                    System.out.println("Hello  "+args[0]);
            }
            else {
                    System.out.println("Hello World ");
            }
   }
}
```

Prog. 11.8.1

This time you will compile the program as you have before (*javac HelloWorld.java*) but you will run it differently. Instead of just entering *java HelloWorld*, use the following command

Java HelloWorld yourName

Of course you would replace *yourName* with your actual first name. Check out the following line of code.

Java HelloWorld yourFirstName yourLastName

If you tried to use this, your last name would be ignored because it would be counted as the second argument to the program (at index 1) and we only use the first argument (at index 0). Arguments passed to the program are separated by spaces. If an integer string is passed as an argument, it will need to be converted before it can be used as an integer. For the sake of simplicity in testing, we will continue to put the main method inside of the same class when we define properties and methods for classes like the Dog class but this isn't the real world programming practice. Classes that are used to define properties and methods of an entity such as the Dog class, and the Calculator, Person, and Car classes (upcoming examples or practice problems) usually don't contain a main method. They are created for other classes to use. So although it is easier to use main methods in these classes for testing purposes, realistically speaking the main method would be in another class that creates objects of the type defined by your "definition" class.

Section 11.9: Getter's and Setters

Up until now, we've made our data fields in the Dog class public. When a method, data field, or constructor is public, it can be accessed within any class. When they are private they can only be accessed within the same class. We will discuss the protected modifier later, but there is one more modifier that hasn't been mentioned. The default modifier is used when no visibility modifier is specified. In this case, the field, method, or constructor is only accessible within the class or within the same package. Packages are used to group related classes. It is not good practice to make data fields public. When you do this, anyone can change your data fields in unpredictable ways. Typically you should make your data fields private and then create methods to access (getters) and change (setters) them. This way, you can define the way a user works with your data fields. The terms getters and setters are just common names for methods that return the value of a particular field (getters) or that takes an argument and sets the value of a particular field (setters). Copy your Dog class to a folder called *11.9.1* and make the necessary changes described below.

```
public class Dog {
    private String color;
    private String breed;
    private int age;
    private String name;
    public Dog() {
            color = "Brown";
            breed = "Mutt";
            age = 1;
            name = "ToTo";
    }
    public Dog(String name, String breed, String color, int age) {
            this.name = name;
                this.color = color;
            this.breed = breed;
            this.age = age;
    }
    public void speak() {
            System.out.println(""+name+ " says Bark!");
        }
        public String getName() {
```

```
    return name;
        }
        public String getBreed() {
    return breed;
        }
        public String getColor() {
    return color;
        }
        public int getAge() {
    return age;
        }
        public void setName(String name) {
    this.name = name;
        }
        public void setBreed(String breed) {
    this.breed = breed;
        }
        public void setColor(String color) {
    this.color = color;
        }
        public void setAge(int age) {
    this.age = age;
        }
    public static void main(String[] args) {
            Dog fido = new Dog("Fido", "German Sheppard", "Gold",3);
            Dog fifi = new Dog("Fifi", "poodle", "white",2);
            fido.speak();
            fifi.speak();
    }
}
```

Prog. 11.9.1

Now if you tried to access one of the properties directly (ex. *fifi.color*) you would

encounter an error. The getters and setters are now the only ways of accessing and setting the fields other than initializing them with the second constructor.

Section 11.10: Overloaded Methods

Just as constructors can be overloaded so can methods. Similar to overloaded constructors, overloaded methods have the same names but a different parameter list. The parameter list can differ in the number, order, or type of parameters, or some combination of these.

Section 11.11: This Keyword

We've talked about the keyword *this* some already. As you know it is used to access data fields of the current class. You can think of *this* as the current object (which is exactly what it is). You may sometimes find it necessary to call one constructor from another constructor in a class. You would use the following syntax to do so

this(optional-parameter-list);

The JVM will know which constructor you are calling based on the parameters passed in.

Section 11.12: Inner Classes

A class can be defined inside of another class or inside of a method (which will itself be inside of a class). These inner classes are the only classes that can have a visibility modifier other than public or default. Defining a class within another class is a security mechanism that insures that only a class' methods and constructors can access and create instances of the inner class. It is the same concept when defining a class inside of a method. That will insure that instances of the class can only be created within this method.

Section 11.13: Null values

A variable that points to an object is a reference variable. It doesn't actually hold the contents of the object but contains an address pointing to the location of the object in memory. The value null can be assigned to a reference variable as a type of place holder to indicate that the variable doesn't currently reference anything but does have a value. When is this useful? Let's say you had created a Scanner object. Then you used an if-statement to determine how the variable was initialized. If an argument was passed in through the command line (like a file name), you chose to use the Scanner to read from the file. Otherwise you read from the command line. Now when the java virtual machine is interpreting the program, it's going to look at your creation of the Scanner reference variable and then look further down and see you using that Scanner variable. The JVM will then come to the conclusion that your variable is being used but was never initialized. This is because the variable is being initialized within an if-statement, which is another scope. The JVM is only smart enough to check the same scope that the reference variable was created in. This is where the null value will come in handy. After (or while) creating the reference variable, we can initialize it to null. Then we can proceed to assign the variable a value based on the condition that we described above. Then we can use it. Now when the JVM checks, it will see that the Scanner variable has been initialized. If you fail to initialize the variable you may or may not get a warning depending on the situation but it is easier to avoid problems this way. Below is the example I described. Write it in a folder called *11.13.1*, compile it, and run it.

```
import java.util.Scanner;
import java.io.File;
public class TryNull {
   public static void main(String[] args) {
                Scanner in = null;
                File f = new File(args[0]);
            if (args.length>0 && f.exists()) {
                    in = new Scanner(f);
                    if (in.hasNext()) {
                            System.out.println("The first line is:
"+in.nextLine());
                    }
                }
            else {
                    in = new Scanner(System.in);
                    String entered = in.next();
```

```
                    System.out.println("You entered "+entered);
                }
    }
}
```

Prog. 11.13.1

Section 11.14: Final Concept Demonstration

This section has seen an introduction of many concepts. This next example seeks to demonstrate many of these concepts as a concise whole. Write, run, and compile the following program in a folder called *11.14.1*.

```
import java.util.Scanner;
public class Calculator {
    private int value1;
    private int value2;

    //constructors
    public Calculator() {
            this.value1 = 1;
            this.value2 = 1;
    }
    public Calculator(int value1,int value2) {
            this.value1 = value1;
            this.value2 = value2;
    }
    //setters
    public void setValue1(int value1) {
            this.value1 = value1;
    }
    public void setValue2(int value2) {
            this.value2 = value2;
    }
    //getters
    public int getValue1() {
            return value1;
    }
    public int getValue2() {
            return value2;
    }
    //add methods
```

```
//adds the class fields value1 and value2
public int add() {
        return value1 + value2;
}
//adds the passed in local variables value1 and value2
public static int add(int value1,int value2) {
        return value1+value2;
}
//subtract methods
//subtracts the class fields value2 from value1
public int subtract() {
        return value1-value2;
}
//subtracts the passed in local variables value2 from value1
public static int subtract(int value1,int value2) {
        return value1-value2;
}
//multiply methods
//mulitplies the class fields value1 and value2
public int multiply() {
        return value1*value2;
}

//mulitplies the passed in local variables value1 and value2
public static int multiply(int value1,int value2) {
        return value1*value2;
}
//divide methods
//divides the class fields value1 by value2
public int divide() {
        if (value2!=0) {
                return value1/value2;
```

```
            }
            else {
                    System.out.println("Division by zero is not valid");
                    return 0;
            }
    }
    //divides the passed in local variables value1 by value2
    public static int divide(int value1,int value2) {
            if (value2!=0) {
                    return value1/value2;
            }
            else {
                    System.out.println("Division by zero is not valid");
                    return 0;
            }
    }
    //main method
    public static void main(String[] args) {
            Scanner in = new Scanner(System.in);
            System.out.print("Enter the first value: ");
            int value1 = in.nextInt();
            System.out.print("Enter the second value: ");
            int value2 = in.nextInt();
            Calculator calc = new Calculator(value1,value2);
            System.out.println("Added: "+calc.add());
            System.out.println("Subtracted: "+calc.subtract());
            System.out.println("Multiplied: "+calc.multiply());
            System.out.println("Divided: "+calc.divide());
            int value3 = 3;
            int value4 = 4;
            System.out.println(""+value4+" + "+value3+" =
"+Calculator.add(4,3));
            System.out.println(""+value4+" - "+value3+" =
```

```
“+Calculator.subtract(4,3));
            System.out.println(””+value4+” * “+value3+” =
“+Calculator.multiply(4,3));
            System.out.println(””+value4+” / “+value3+” =
“+Calculator.divide(4,3));
   }
}
```

Prog. 11.14.1

Notice the getters and setters for the data fields. Notice that we have two overloaded constructors. Note the fact that each mathematical method as two overloaded versions. One version is an instance method that operates on a specific object and can therefore access and/or manipulate the data fields of the current object. The other is a static method that can only operate on static data fields which aren’t attached to a specific object, and local variables. The static math methods would be useful if you didn’t want to bother creating a calculator object for whatever reason. Finally notice that you call a static method using the class name instead of the object name, as mentioned before.

Practice Problems

1. Create a Car class with the properties color (*String*), year (*int*), miles (*int*), make (*String*), and model (*String*), all of the proper getters and setters, a no argument constructor that initializes the fields to some default values, a five argument constructor which takes all five properties as arguments and sets the data fields according to these values. There should also be a main method that creates two car objects one using the first constructor and then setting the values using the setters, and one using the second constructor, and then finally uses the getters to print out the properties of both objects.
2. Create a Person class with the properties *firstname*, *lastname*, *street*, *city*, *state*, *zip*, *country*, all strings, a two argument constructor that takes the first name and last name of the person and sets them, and all of the proper getters and setters. There should also be a main method that creates a person, uses the setters to set the properties, and uses the getters to print the properties out.

Solutions to these problems can be found on the website (www.jtjackson.org).

Chapter 12: Exceptions

Section 12.1: What are Exceptions and Why do They Occur

An exception is a problem during the execution of a program that interrupts the programs normal flow and causes it to terminate abnormally. This abnormal termination is not recommended, which is why exceptions should be handled.

Exceptions can occur for many reasons. They can result from user error such as a user entering invalid data, from programmer error such as an attempt to open a file that can’t be found, or from system failures such as the JVM running out of memory. There are three categories of exceptions, which are, checked exceptions, unchecked exceptions, and error.

Section 12.2: Types of Exceptions

Checked exceptions are exceptions that occur at compile time. The JVM requires the programmer to handle these exceptions. For example, if you created a *FileReader* object to read from the file ***temp.txt*** that you created back in Section 10.3 as demonstrated below.

```
import java.io.File;
import java.io.FileReader;
public class FileNF {
   public static void main(String[] args) {
            File file = new File("temp.txt");
            FileReader filereader = new FileReader(file);
   }
}
```

Prog. 12.2.1

Attempt to compile this program. You will get a message about and an unhandled exception. The exception that you failed to handle was a file not found exception. This means that you didn't handle the possibility that a file not found exception could occur. As you can see, failing to handle a checked exception that could arise in you program will cause an exception.

Unchecked, or runtime, exceptions occur at runtime and are ignored during compilation. The programmer is not required to handle these exceptions but it's important to try to write your code so that these kinds of exceptions can be avoided. An example of a runtime exception is if your program uses an invalid index in an attempt to access an element in an array. Another example is if a program attempts to divide a number by zero. Say both of these programs were receiving input from a user that caused these exceptions to occur. A good way to program around situations like this would be to use if statements to check the validity of values before using them.

Errors are not really exceptions at all but are problems that are beyond the control of the user or programmer. You usually won't account for the potential of errors in a program because there isn't much that you can do about them. An example, mentioned earlier, is if the system runs out of memory. There are a couple of ways to handle exceptions in java. You can either catch them or throw them.

Section 12.3: Catching Exceptions

To catch a possible exception you would surround the code that could potentially cause an exception with a try-catch statement. The potentially error causing code would go in the try-block and the code that should be run in the event of an error would go in the catch-block. There is also an optional finally-block. When a finally-block is included, the code within it always runs, whether an exception occurs or not. It is useful for cleanup code that you want to happen no matter what. Copy your *FileNF* program a folder called *12.3.1* for this section and make the necessary changes.

```
import java.io.File;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;
public class FileNF {
        public static void main(String[] args) {
                File file = new File("temp.txt");
          FileReader filereader = null;

          try {
                filereader = new FileReader(file);
                filereader.close();
          }
          catch (FileNotFoundException fnf) {
                System.out.println("The file "+file.getPath()+" was not found
      ");
          }
          catch (IOException ioe) {
                        System.out.println("An Input\Output exception
      occured");
          }
        }
}
```

Prog. 12.3.1

Now compile and run the program to see what I mean. You shouldn't have any problems.

On your first run, don’t copy the temp.txt file into your directory so you can see how the try-catch works. Then you can copy the file and test again. Notice the catch block. It’s similar to the definition of a method in that you define the type of object that it will catch and then give the parameter a name. Of course a catch block can only catch Exceptions and Errors. You already know that the *FileNotFoundException* and *IOException* classes need to be imported. So did the *File* and *FileReader* classes. Since all of the imports come from the *java.io package*, I could have simply used the import statement *import java.io.** to get all of the classes in this package but I prefer to only import what I need.

A try-catch statement can have one or more catch blocks as demonstrated above. If instead, you want to perform the same tasks in the event of several different exceptions, you can use the following syntax for you try catch block.

```
try {
   //some code
}
catch (Exception1|Exception2|Exception3 ex) {
   //catch-block
}
```

Ex. 12.3.1

Section 12.4: Throwing Exceptions

If a method doesn't handle a checked exception, it must throw it. This means that the method passes the exception to the method that called the current method, and requires that method to handle the exception. Obviously the main method cannot throw exceptions because it doesn't have a calling method. In order for a method to throw a certain type of exception, it must declare that it throws an exception of this type by following it's signature with the *throws* keyword and the exception type. Write the following example in a folder called 12.4.2 and compile and run it.

```
import java.io.*;
public class Thrower {
    public static void main(String[] args) {
            try {
                    Thrower.divide(3,0);
            }
            catch (ArithmeticException ae) {
                    System.out.println("An arithmetic exception has occured");
            }
    }
    public static int divide(int x,int y) throws ArithmeticException {
            int value = 0;
            value = x/y;
            return value;
    }
}
```

Prog. 12.4.1

As you can see in the above example, the divide method throws the exception to the main method which then handles the exception. Allowing a method to throw exceptions can give a programmer who is using your class more control because they can decide what their program should do in the event of an issue. We could have also chosen to stop the program in the divide method when an exception was encountered, but this would take the control away from the programmer using your class. They may not want their program to stop, but instead to recover. By the way, the error occurred because we attempted to divide by zero.

Exceptions and Errors are just another type of object. This means that you can create an Exception or error object the same way that you would create any other object (using the new keyword). With that being said, instead of immediately throwing the

exception, you can throw an exception that you created. Copy your last program to a folder called *12.4.2* and make the necessary changes to get the program below.

```
import java.io.*;
public class Thrower {
    public static void main(String[] args) {
            try {
                    Thrower.divide(3,0);
            }
            catch (ArithmeticException ae) {
                    System.out.println("An arithmetic exception has occured");
            }
    }
    public static void divide(int x, int y) throws ArithmeticException {
            int value = 0;
            if (y!=0) {
                    value = x/y;
                    return value;
            }
            else {
                    throw new ArithmeticException("Division by zero is
invalid");
            }
    }
}
```

Prog. 12.4.2

You could have used the no argument constructor to create the arithmetic exception but it can be more useful to use a custom error message, so I chose the constructor that accepts a string and sets the error message with it. As far as I know, all java exception classes include a constructor of this form but when in doubt, google search a particular java class and you'll find plenty of information about it. Something else you could have done is caught the exception using a try-catch statement in the divide method and then thrown it to the main method.

We haven't talked about subclasses and subtypes yet but (that's chapter 14) but if a catch block catches a certain type of exception it can also catch any subtypes of that

type. So if you wanted your catch block to catch all Exceptions, you would just do something like this.

```
        try {
    //some code
        }
        catch (Exception e) {
    //some code
        }
```

This will work because all exceptions are subclasses of the exception class. You'll need to import the Exception class.

Section 12.5: Exception Built in Methods

Just like all other java classes, the Exception classes have built-in methods. Three important ones, in my opinion, are *getMessage*, *toString*, and *printStackTrace*. The *getMessage* method returns a detailed message about the exception. The *getCause* method returns a string representation of the exception which includes the name of the class and the result of *getMessage*. The *printStackTrace* method prints the result of the *toString* method along with the stack trace of *System.err*. There are other methods of course, but these are the ones that I believe you'll use the most.

Chapter 13: Scope

Now it's time to get back to the concept of scope that I mentioned earlier. Scope defines the block in which a variable or field is accessible. For example, if a variable or inner class were defined in a class, they are accessible anywhere in that class. Therefore they have a global scope. On the other hand, if a variable is defined within a method, constructor, if-statement, while-loop, or for-loop, it will only be accessible within that block, from the point of declaration on down. This variable, then, has a local scope. I want to explicitly point out that each set of curly braces denotes a new scope. Write the following program in a folder called *13.1*, compile it, and run it.

```
public class Messenger {
    private String message;
    public Messenger() {
            message = "Hello World";
    }
    public Messenger(String msg) {
            setMessage(msg);
    }
    public void setMessage(String msg) {
            message = msg;
    }
    public String getMessage() {
            return message;
    }
    public static void main(String[] args) {
            Messenger m = new Messenger("I love lucy!");
            int x = 10;
            for (int i = 0; i<x; i++) {
                    int j = 1;
                    System.out.println(m.getMessage());
            }
    }
}
```

Prog. 13.1

You see the message data field? This is a global variable. It can be accessed all throughout the class. The order of class variables, constructor and method declarations doesn't matter but I prefer to define all of my class variables at the top of the program. Ordering is relevant in some other programming languages. Please note that the message variable couldn't be accessed in the main method without creating an instance of the messenger class first. This is because the main method is a static method that doesn't work with any specific instance of the class and the message variable is an instance variable that must belong to a specific instance of a class. On the other hand, the getter and setter methods are perfectly able to access the messenger variable because they are instance methods that do work with specific instances of the class. The variable x is a local variable. It is local to the main method and cannot be accessed outside of it, or in any code in the main method that came before its declaration. The integer's *i* and *j* are both local to the for-loop and any attempt to use them outside of this loop would cause an error. The variable *x* is accessible anywhere within the for-loop, including the signature, because it was declared first, outside of the loop.

Chapter 14: Inheritance

Section 14.1: Inheritance Explained

Inheritance is an important concept in java, and in any object oriented language. Inheritance is the process by which a subclass inherits the properties and methods of a superclass. In order to accomplish this, the subclass extends the superclass. For example, let's say you had an animal class. The animal class had the properties *species*, *habitat*, and *sex*. This class could then be extended by subclasses called *Dog* (like the one you created), *cat*, *squirrel*, *elephant*, *zebra*, etc. These subclasses would inherit all non-private fields and methods of the animal class. If you want to take it a step farther, you could choose to extend the dog class again with the class' *wolf*, *coyote*, and *housepet*. You could also choose to extend the cat class farther with the classes *cheetah*, *lion*, *tiger*, *domesticated* (house pet), *liger* and *tygon* (liger and tygon are actually things, google them).

Section 14.2: Inheritance Extended Example

Here's an example of a person class developed for a school and extended by the classes Student and Teacher. If you worked on the first practice problem in chapter 11, then you can copy your program to a folder called *14.2.1* and make the necessary changes to get the Person class described below. All classes are in separate files.

```
import java.util.Date;
import java.util.Calendar;
public class Person {
    private String fname;
    private String lname;
    private String[] address;
    private Date birthday;
    public Person(String fname, String lname) {
            //address[0] = street, address[1] = city, address[2] = state,
                //address[3] = zip, address[4] = country
            address = new String[5];
            birthday = new Date();
            this.fname = fname;
            this.lname = lname;
    }
    public void setBday(int month, int day, int year) {
            Calendar c = Calendar.getInstance();
            c.set(1974,1,15); //Set the date of the Calendar to Feb 2, 1974
            birthday = c.getTime();
    }
    public Date getBday() {
            return this.birthday;
    }
    public void setFname(String fname) {
            this.fname = fname;
    }
    public String getFname() {
            return fname;
```

```
    }
    public void setLname(String lname) {
            this.lname = lname;
    }
    public String getLname() {
            return lname;
    }
    public String getName() {
            return fname+" "+lname;
    }
    public String getAddress() {
            String addr = "";
            for (int i=0; i<address.length; i++) {
                    addr  += address[i];
                    if (i!=(address.length-1)) addr += ",";
            }
            return addr;
    }
    public void setAddress(String street, String city, String state, String zip, String
country) {
            address[0] = street;
            address[1] = city;
            address[2] = state;
            address[3] = zip;
            address[4] = country;
    }
}
```

Prog. 14.2.1a

The Person class is the superclass. As you will see soon, the Teacher and Student classes use the keyword *extends* to extend the Person class. A class cannot extend more than one class. The Person class has the fields named *fname*, *lname*, *birthday*, and *address*. I chose to use an array of Strings to represent the different components of an address.

The birthday is a date object. You haven't used the date class before. There is more than one date constructor but I chose to use the no-argument constructor to create the

date. By default, when a Date object is created, it represents the current date and time. The Date class has many getters and setters but most of them are deprecated meaning out of date or going out of date, so I used a Calendar object to change the birthday field in our *setBday* method. You also haven't used a Calendar object yet. It is a class with properties such as month, day, year, time, etc. that represent a Calendar. To get an instance of Calendar use *Calendar.getInstance*. Notice that *getInstance* must be a static method since it is called using the class name instead of requiring you to create an object and call the method on that.

Then the calendar *set* method is used to set the calendar date to February 15, 1974. The arguments are in the order year, month, and day. Notice that we use 1 for February instead of 2. That is because the months are represented using integers 0-11. An interesting side note is that if you try to use the integer 12 as the month value (say for the date December 15, 1974), the calendar will flip to the next month and next year and give you the date January 15, 1975 instead.

Although I have getters for both *fname* and *lname*, I also decided to create a getter than returns a string that contains the *fname* followed by the *lname*. In the *getAddress* method I iterated through the address array, adding each index to the *addr* string. As *long* as the for-loop wasn't on the last element in the array, I also added a comma to the end to separate the current value and the next value. Notice that the += operator can also be used in the concatenation of strings. Plus (+) and plus equal (+=) are the only arithmetic operators that can be used with strings. Everything else in the code is pretty self explanatory. Now create the teacher class in the same folder.

```
public class Teacher extends Person {
    private String tid;
    private int salary;
    private int years_of_emp;

    public Teacher(String fname,String lname,String tid) {
            super(fname,lname);
            this.tid = tid;
            this.years_of_emp = years_of_emp;
            this.salary = salary;
    }
    public void setSalary(int salary) {
            this.salary = salary;
    }
    public int getSalary() {
```

```
        return salary;
    }
    public void setYears(int years) {
        this.years_of_emp = years;
    }
    public int getYears() {
        return years_of_emp;
    }
    public void setTid(String tid) {
        this.tid = tid;
    }
    public String getTid() {
        return tid;
    }
}
```

Prog. 14.2.1b

The teacher class has three properties, *tid*, *salary*, and *years_of_emp*. It also has the basic getters and setters. Look at the constructor. It takes the first name and last name as arguments. Then it passes the values to what looks like a method name *super*. This is actually a call to the parent constructor. Super will call whatever constructor has the correct parameter list, based off of the passed in values. This is necessary in order to create an instance of the parent object, when an instance of this object is created. The keyword super can also be used to access non-private methods and properties in the superclass. For example

```
super.getName();
```

Properties of the parent class can be accessed using the following syntax, assuming the values aren't private.

```
super.propertyName
```

Super can be thought of as the object representing the parent class, that was created during instantiation of this object. Instantiation is the creation of an object using its constructor. Next, create the Student class, in the same folder.

```
public class Student extends Person {
    private String sid;
    private int grade_level;
```

```
public Student(String fname,String lname,String sid,int grade) {
        super(fname,lname);
        this.sid = sid;
        setGrade(grade);
}
public void setSid(String sid) {
        this.sid = sid;
}
public String getSid() {
        return sid;
}
public int getGrade() {
        return grade_level;
}
public String getGradeStr() {
        if (grade_level == 0) {
                return “Kindergarten”;
        }
        else if (grade_level == 1) {
                return “1st”;
        }
        else if (grade_level == 2) {
                return “2nd”;
        }
        else if (grade_level == 3) {
                return “3rd”;
        }
        else {
                String gradestr = ””+grade_level;
                return gradestr+=“th”;
        }
```

```
    }
    public void setGrade(int grade) {
            if (grade>=0 && grade<13) {
                    this.grade_level = grade;
            }
            else {
                    grade_level = 9;
            }
    }
}
```

Prog. 14.2.1c

The Student class has two properties, *sid*, and *grade_level*. It also has all of the basic getters and setters. You'll notice the call to the super constructor in the constructor. There's also a method that returns a string representation of the grade of a student's grade level. Of course the Student and Teacher classes could technically have more properties. The Student could have arrays of completed classes, current classes, and grades. The teacher could have arrays of the classes they are teaching and their students. For the sake of saving space, and not overcomplicating the lesson, I chose to leave those things out. Finally, create a tester class to test these classes, in the same folder, compile it, and run it.

```
public class Tester {
    public static void main(String[] args) {
            Student s = new Student("Michael","Hunt","s456",10);
            Teacher t = new Teacher("Carie","Johnson","t789");
            t.setSalary(45000);
            t.setYears(5);
            System.out.println("The student "+s.getName()+", student number "+s.getSid()+", is a "+s.getGradeStr()+" grade student at our school");
            System.out.println("The teacher "+t.getName()+", teacher number "+t.getTid()+", has been working with us for "+t.getYears()+" years and makes a salary of $"+t.getSalary()+" a year");
    }
}
```

Prog. 14.2.1d

Now I want to talk a little about the protected visibility modifier. When a field is declared as protected, it is accessible to all classes within the same package. The field can also be

accessed outside of the class, but only by subclasses of the class where the field was defined. Feel free to play around with these examples, adding and/or taking away certain properties, testing some of the other methods, and playing with the visibility of different fields to see what happens.

Section 14.3: Method Overriding

We've already talked about constructor and method overloading which is when you have two constructors or methods in the same class, with the same name, and a different parameter list. People often confuse Overriding and Overloading but they are different. Method Overriding is when a subclass essentially rewrites the functionality of a superclass method in order to make its functionality more specific to the subclass. An overridden method should have the same argument list as the original method and should return the same type as the original method or a subtype of the original type. Instance methods can only be overridden if they are inherited (so in other words, not private instance methods). Final methods (which we won't discuss) can't be overridden. Static methods can't be overridden but can be re-declared. This means that when working with the current class, the superclass static method that you re-wrote will for all intents and purposes be non-existent. You won't be able to access it using the super keyword. Here's a simple example of an Animal class extended by a Cow class that overrides the Animal class' *speak* method.

```
public class Animal {
    //I chose not to define a constructor and just use the default constructor
    public String speak() {
            return "All animals make noise";
    }
}
```

Prog. 14.3.1a

```
public class Cow extends Animal {
    public Cow() {
            super();
    }
    public String speak() {
            return super.speak() + " but Cows say moo!";
    }
    public static void main(String[] args) {
            Cow cow = new Cow();
            System.out.println(cow.speak());
    }
}
```

Prog. 14.3.1b

Compile and run the Cow program. Method overriding allows you to define general functionality for all animals and change and/or extend that functionality in subclasses of the animal class.

Section 14.4: The Object Class

We've just finished creating a parent class (superclass) and two child classes (subclasses). The child classes Teacher and Student are subtypes of the Super type Person. Inheritance can go on indefinitely. A parent class can have several child classes which each has several child classes and so on. All classes are children of the Object Class. This is the top superclass with no class above it. *String*s, *File*s, *Scanner*s, *PrintWriter*s, *Exception*s, arrays are all Objects along with any class types you've created. All objects implement the methods of this class. There are two methods in the Object class that I want to talk about. That is the *equals* method and the *toString* method.

You may remember us using the *equals* method on a *String* object. This method is in the *String* class and it overrides the *equals* method in the object class. Most if not all objects should override this method to allow the comparison of objects of this type. Copy your Dog class from section 11.9 to a folder called *14.4.1*. Then add the following code.

```
    @Override
    public boolean equals(Object o) {
if (o instanceof Dog) {
        int age = ((Dog)o).getAge();
                String breed = ((Dog)o).getBreed();
                String color = ((Dog)o).getColor();
                String name = ((Dog)o).getName();
        if (this.getAge()==age &&
                this.getBreed().equals(breed) &&
                this.getColor().equals(color) &&
                this.getName().equals(name)) {
                return true;
        }
}
return false;
    }
```

Also add these three lines to the main method.

```
Dog fido2 = new Dog("Fido","German Sheppard","Gold",3);
System.out.println("Are fido and fido2 equal? "+fido.equals(fido2));
System.out.println("Are fifi and fido equal? "+fido.equals(fifi));
```

Compile and run the new program. Notice the *@Override* tag. We hadn't used this in the

previous method overriding example. What it does is explicitly tell the JVM that you are attempting to override a method. So if for whatever reason you're not actually overriding a method, like if you used a different name, or an incompatible return type or the wrong parameter list, you will get an error. If you don't use this tag, you could think you were overriding a method and are actually creating a new one. As you can see, an overridden *equals* method should always accept an Object and return a *boolean*. Then you would use the *instanceof* keyword to determine if the Object is of the correct type. We'll discuss the *instanceof* keyword more in the next section. In the above equals method, if the Object is an instance of the Dog class and all of the properties are equal to the current object, true will be returned, thus ending the method and never making it to the return false statement. Otherwise the if-statement will complete and at the end of the method, false will be returned.

The toString method is a method in the Object class that returns a string representation of an object. In the Object class the method returns a string with information about the object's location in memory. It's not a very user friendly string. That's why many classes choose to override this method to return specific information about an object. Add the following code to your dog class.

```
@Override
public String toString() {
        String desc = this.getName()+" is a "+this.getAge()
                +" year old, "+this.getColor()+", "+this.getBreed();
        return desc;
}
```

Also add the following lines of code to the main method.

```
System.out.println(fido.toString());
System.out.println(fifi.toString());
```

Run and Compile the code.

Chapter 15: Polymorphism

Polymorphism in java, or any object oriented language, is the ability of an object to take on many different forms. Most commonly, this is demonstrated when a parent class reference variable is used to reference a child class object. Assume that we have the Animal class and its subclass Cow described in chapter 14. The following example demonstrates polymorphism.

Animal a = new Cow();

This can be an important mechanism when programming. Say your Animal class had a couple other subclasses such as dog and cat. Now say you wanted to have an array to hold cows, dogs, and cats. We already know that an array can only hold one type of value. So by declaring the array as type Animal as follows, you can hold all three.

Animal[] animals = new Animal[5];

The same concept holds when you want to create a method that takes as a parameter and/or returns dog, cat, and cow objects. Declaring the return type or the specific parameter as a variable of type Animal, you can do just that. You could also write three overloaded methods, but depending on what you're trying to accomplish, that route may be more complicated. So what if your method took an Animal as a parameter. How would you know if you received a Dog, Cat, or, Cow object? You would use the *instanceof* keyword. See the following example for a demonstration

```
public void whatIsIt(Animal a) {
    if (a instanceof Dog) System.out.println("Dog");
    else if (a instanceof Cat) System.out.println("Cat");
    else if (a instanceof Cow) System.out.println("Cow");
    else System.out.println("It's just a plain old animal");
}
```

Ex. 15.1

Whether the argument is a Cat, Dog, Cow, or Animal object, using *instanceof* Animal will always return true because all of these objects are Animal objects. That is how polymorphism works. Seeing that if the argument isn't a Cat, Dog or Cow it can only be an Animal object (assuming that Cat, Dog, and Cow are the only subtypes of Animal) I placed the plain old animal print statement as the option if all else failed.

Chapter 16: Abstract Classes

An abstract class is something like a template. An abstract class allows you to define the structure of a class while leaving the implementation details for later. An abstract class may or may not have abstract methods but if a class contains one or more abstract methods, it must be an abstract class. In order to make use of an abstract class the programmer has to extend it to inherit its properties and then provide implementations for the abstract methods. Abstract classes cannot be instantiated, or in other words, you can't create an instance of an abstract class. The only time that a class inheriting from an abstract class isn't required to provide implementations for the abstract methods, is when the class itself is abstract. The rules of inheritance of an abstract class's properties and methods are the same as with any other class. Here's an example of an abstract class using the earlier mentioned Animal class which has the subclasses Dog (which was never actually created to extend the Animal class), Cat (which was never actually created), and Cow.

```
public abstract class Animal {
    public abstract void Speak();
}
```

Ex. 16.1

Notice that the abstract keyword follows the visibility modifier when declaring abstract classes and methods (when a visibility modifier is used, that is). Cow, Dog, and Cat would still extend the class as they had done before. I chose to change the Speak method to an abstract method and require the child classes to implement it as they saw fit. This ensures that any non-abstract child classes (even if the first non-abstract child class is a few generations down) must implement the method. This was a choice of design though, and wasn't required. As I've already pointed out, an abstract class doesn't have to have any abstract methods. Notice that when declaring an abstract method, the curly braces where the actual code of the method is defined, disappear. That is because implementation of an abstract method is left to non-abstract subclasses of an abstract class.

Chapter 17: Interfaces

An interface is similar to a class in some ways but there are some important differences. Interfaces are declared in a java file in the same way that classes are. While a class defines the properties and behaviors of an object, an interface describes behaviors for a class to implement. All methods in an interface are abstract and therefore the abstract keyword is not used. An interface can't be instantiated and therefore doesn't have constructors. The only fields that an interface can contain are declared both static and final. A class does not extend an interface but instead implements it. Although a class can only extend one class, it can implement many interfaces, even while extending a class. Let's say you had the classes Person, Dog, and Cheetah. These species don't have very much in common but they do all eat, run, and drink. Depending on what the programmer is trying to accomplish, they may choose to create an Animal interface that defines all of their shared behaviors. Here's a condensed example below

```
interface Animal {
    public void eat();
    Public void drink();
    Public void run();
}
public class Dog implements Animal {
    //Assume the code is here
}
```

Ex. 17.1

Implementing an interface is equivalent to signing a contract which requires that you define all of the behaviors in the interface. If a class doesn't define all of the behaviors in an interface that it implements, then it must be an abstract class.

When overriding an interface method the same signature (name and parameter list) should be maintained as well as the same return type or a subtype of that type.

Note that an interface can extend another interface in the same way that a class can extend another class. Other than the constants (final variables), and abstract methods, an interface may also contain static methods, inner classes, and default methods (not discussed here). The static methods and default methods are the only methods allowed to have a body.

Closing Remarks

We've discussed a lot in a *short* amount of time (hopefully). I truly hope that I've given you the tools to continuing learning and writing java code. I realize that what I think may be a reasonable explanation of a topic may not always be clear to someone else and for that reason I hope that you will contact me with all questions and/or concerns. You can find my up to date contact information on my website at www.jtjackson.org. You can use this information to contact me directly or you can use the discussion board on the site to pose questions directly to myself and fellow readers of the book. I'll try to regularly keep an eye on the discussion board to provide answers to questions and confirm/extend the information in someone else's response. If you encounter any issues with the programs in this book or typos, please let me know. You will also be able to download the example problems and solutions to the practice problems on the website. Every time that I felt that I had finished the book there was more information I wanted to add until I eventually had to remind myself that this was meant to be a *short* guide at which point I made the decision to leave certain things out. I really hope you enjoyed the book and I would love your feedback. Thank you for giving it a try.