

package com.howtoprogramwithjava.io;

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
```

public class FileIO

```
{
    public FileIO (String filename)
    {
        BufferedReader br;
        try
        {
            br = new BufferedReader(new FileReader(filename));
            String line = "";
            while((line = br.readLine()) != null)
            {
                System.out.println("Reading Line: " + line);
            }
        } catch (FileNotFoundException e)
        {
            System.out.println("There was an exception! The file was not found!");
        } catch (IOException e)
        {
            System.out.println("There was an exception handling the file!");
        }
    }
}
```

2nd Edition

TREVOR J. PAGE

Contents

Legal Junk.....	9
Links	10
Audio.....	10
Video.....	10
Chapter 1	11
Variables	12
Control Structures.....	15
Data Structures	18
Syntax	21
Tools	24
Chapter 2	28
The Java Hello World Program.....	29
Java Hello World (part II)	35
Chapter 3	38
What is a Method?.....	39
Object Oriented Programming in Java	42
Java Arrays	45
What are Java Arrays?.....	48
Primitive Data Types	51
What do primitive data types look like in Java?.....	51
Strings	54
String – Object or Primitive?	54
String Concatenation	55
Packages	57
What is a Java Package?.....	57
Why are Java Packages Useful?.....	57
What do Packages look like in Code?	57
What do Java packages look like in code?.....	58
Imports	60
What are Java Imports?	60
What do Java imports look like in Code?	60

Why do I need Java Imports?	61
Is there an easy way to use Java Imports?	61
Importing ALL Classes in a Package	62
One final note	62
Chapter 4	65
Java Object.....	66
Everything is an Object	66
Seriously, is everything an Object?	67
What are the implications to inheriting from Object?	68
How do we override the equals() method?	70
<i>Loose ends</i>	73
Summary.....	74
What is Java Inheritance?	75
How does Inheritance help us?.....	75
What is a super Class and what is a child Class?	75
Coding Inheritance in Java	76
Interface.....	76
Abstract Class.....	78
The Static Keyword	84
What does Static mean in Java?.....	84
Constructors	87
Why are Constructors important?	87
What do Constructors look like in Code?	87
This Keyword.....	91
Why would an Object refer to itself?	91
Access Modifiers	94
What are Java Modifiers?	94
Examples of Access Modifiers in Java:	94
Objects and Methods.....	97
Coding Exercise	98
Let's summarize	98
What is a Constant in Java?	99

The final Keyword	100
private vs public Constants	100
Chapter 5	103
Exceptions.....	104
What are Exceptions in Java?.....	104
Exception Handling in Java.....	104
More details about Exceptions in Java	106
What happens if you don't use a try/catch?	106
When should I use Exceptions?.....	108
Bonus material.....	109
Final words on Exceptions.....	110
String Manipulation	111
What is String Manipulation?.....	111
Searching Strings.....	112
Matching Strings	113
Substring	114
Try it Yourself.....	114
Casting	115
What is Casting in Java?.....	115
What are the Rules behind Casting Variables?	115
What are the benefits of casting variables?.....	116
What are the downsides to Casting?	116
Summary.....	117
Loops	118
What kinds of Loops are there in Java?.....	118
Bonus Content	121
Summary.....	122
Java UI.....	124
First of all, what is a User Interface?	124
Why do I think Java UI is weak?	124
What UI should I use then?	125
Operators.....	126

What is a Java Operator?	126
Equality and Relational Operators	126
Conditional Operators.....	127
Ternary Operator (also a Conditional operator)	128
Arithmetic Operators	129
Remainder Operator	130
Unary Operators	131
Summary.....	132
Enums	133
Nested IF Statements.....	45
The nested IF.....	45
Summary.....	47
Chapter 6	141
Java Multithreading	142
What is it?	142
How do we use it in Java?	142
Pitfalls of Java Multithreading	145
Singleton Design Pattern.....	146
What the heck is a Design Pattern?	146
Enter the Singleton Design Pattern	146
Show me an example!.....	147
Java Recursion	153
So why use Java Recursion?	154
The Big Question.....	155
Mastering Regular Expressions	161
What is a Regular Expression?	161
So how is Regex different from using the <code>indexOf()</code> method?.....	161
Metacharacters – AKA Wildcards.....	162
Regex Ranges	163
Predefined Character Classes.....	164
Time for You to Start Mastering Regular Expressions	164
Sorting Collections in Java.....	166

Comparator/Comparable Interfaces	166
How Would You Solve this Problem?	166
Let's Solve our Problem	167
Comparator's <code>compare(Object o1, Object o2)</code> Method	169
Chapter 7	174
Java Serialization.....	175
What is Java Serialization?.....	175
Powerful stuff people	176
Full Example.....	177
Now it's your turn	178
Connecting to a Database	179
What is a Database?	179
What is a Database Management System?.....	180
What Database Operations can I Perform in Java?	180
Let's see some code	181
Common Pitfalls when Setting up a Database Connection	187
Still Can't Figure it Out?	188
Chapter 8	189
Getting Stuck on a Programming Problem	190
How to Program with Java.com	190
Stackoverflow.com	190
Google.....	191
Trial and Error	193
Step Away from the Problem!.....	193
What is Unit Testing?	195
The Application Code	196
JUnit.....	197
How to Run the Test	199
Negative Tests.....	200
Summary.....	202
Don't Be Clever	203
This code is too complicated!	203

Keep It Simple Stupid (KISS)	203
How do I Keep it Simple Stupid?	204
Refactoring Tools	206
What is Refactoring?	206
Refactoring Tools	206
How to extract Local Variables.....	207
How to Extract a Method.....	208
Rename Variables	210
How to Change a Method's Signature	210
Summary.....	212
Challenge Yourself	213
Continuous Education	213
The Best Decision I Made.....	213
Adapt to Changing Times	214
Serendipity.....	214
Summary.....	215
Test Driven Development	216
So what is Test Driven Development?.....	216
Okay so this sounds great, but how can I start?.....	217
Bonus Content Chapter 1.....	224
What is a Web Application?	225
Why are Web Applications so Great?	225
Global Access to Application	225
Instant Delivery of Bug Fixes	225
Cross Platform Capability.....	225
What Technologies are used when building Web Applications?	226
Spring Framework.....	226
Hibernate	228
MySQL.....	229
HTML.....	229
JavaScript	229
JSPs (JavaServer Pages)	230

Bonus Content Chapter 2.....	231
Getting Started	232
Install Springsource Tool Suite and JDK	232
Install MySQL Server and TOAD	232
Bonus Content Chapter 3.....	235
The Secret Sauce.....	236

Legal Junk

Copyright © 2013 by Ecosim Software Inc.

All rights reserved. No part of the contents of this eBook may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the author, EcoSim Software Inc., nor its resellers or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

How to Use this eBook

Links

There are links to websites throughout this eBook, feel free to treat them like any other link you would find on the internet. When you click on them, you will be taken to a website which may not be associated to this eBook or its author.

Audio

Throughout this eBook, there are audio guides that accompany certain lessons and can be used as an addition to the written content. These audio segments are pulled from the “How to Program with Java” podcast, and if you really enjoy listening, you can feel free to [subscribe to the podcast here](#).



Video

As an added bonus, there are now links to video segments provided in this eBook. The links will take you to a YouTube video that can be used as a valuable learning tool.



Full Length Video Tutorials

If your goal is to learn Java quickly, I would also suggest you check out the [Java Video Tutorials](#) course that I teach online, which supplements this eBook nicely. It's a great place to learn Java and ask questions when you get stuck, as there's an entire community of Java programmers that are learning with you that will be able to answer your questions!

Chapter 1

The 5 Basic Concepts of any Programming Language

Variables



I want this content to provide anyone “walking in off the street” the knowledge to be able to write their first program with the Java programming language with as little pain as possible.

So, let’s get started with our first topic: The 5 basic concepts of any programming language. You might say, “Why are we talking about any programming language? I thought this was about Java.” Well, I’ve found that it’s important to remember that a lot of programming languages are very similar and knowing what’s common between all programming languages will help you transition into any other programming language if you need to! For example, with the Java programming knowledge I had obtained, it took me less than a month to learn how to program in a language called Objective C (which is used for iPhone apps). That’s powerful stuff!

So, here are the **5 basic concepts of any programming language**:

1. **Variables**
2. *Control Structures*
3. *Data Structures*
4. *Syntax*
5. *Tools*

I recognize that these words probably look foreign to you, but don’t worry; I’ll do my very best at taking the mystery out of them. Now, there’s a lot to say about each of these 5 concepts, so I’ll only talk about item #1, **variables**!

What is a variable?

Variables are the backbone of any program, and thus the backbone of any programming language. Wiki defines a variable as follows:

In computer programming, a variable is a storage location and an associated symbolic name which contains some known or unknown quantity or information, a value.

Okay, well, that’s kind of cryptic. To me, a variable is simply a way to store some sort of information for later use, and we can retrieve this information by referring to a “word” that will describe this information. For example, let’s say you come to my website [www.howtoprogramwithjava.com](http://howtoprogramwithjava.com) and the

first thing I want to do is ask you what your name is, so I can greet you in a nice way the next time you visit my website. I would put a little text box on the screen that asks you what your name is... that text box would represent a **variable**! Let's say I called that text box "your Name." That would be the symbolic name (or "word") for your variable (as described from our wiki definition above). So now, when you type your name into the text box, that information will be stored in a variable called "your Name." I would then be able to come back and say "What value does the **variable** 'your Name' contain?" The program will tell me whatever it was you typed into that text box.

This concept is extremely powerful in programming and is used constantly. It is what makes Facebook and Twitter work, it's what makes paying your bills via your online bank work, and it's what allows you to place a bid on eBay. Variables make the programming world go around.

Now, if we want to get more specific, when it comes to the Java programming language, variables have different **types**. *Brace yourself here, for I'm going to try to confuse you by explaining an important concept in three sentences.* If I were to be storing your name in a variable, that **type** would be a **String**. Or, let's say I also wanted to store your age; then that **type** would be stored as an **Integer**. Or, let's say I wanted to store how much money you make in a year; then that **type** would be stored as a **Double**.

What the heck are **String**, **Integer** and **Double**?

Excellent question! In Java, the programming language wants to know **what kind of information** you are going to be **storing in a variable**. This is because Java is a **strongly typed language**. I could teach you about what the difference is between a strongly typed language and a weakly typed language, but that will likely bore you right now, so let's just focus on **what a type is in Java and why it's important**.

Typing in Java allows the programming language to know with absolute certainty that the information being stored in a variable will be "a certain way." So like I said, if you're storing your age, you would use the **Integer** type and that's because in Java an **Integer** means you have a number that won't have any decimal places in it. It will be a whole number like 5, 20, 60, -60, 4000 or -16000. All of those numbers would be considered an **Integer** in Java. What would happen if you tried to store something that wasn't an Integer into an Integer variable like the value "\$35.38"? Well, quite simply, you would get an error in the program and you would have to fix it! \$35.38 has a dollar sign (\$) in it as well as a decimal place with two digits of accuracy. In Java, when you specify that a variable is a type Integer, you are simply not allowed to store anything except a whole number.

Now, I don't want to go into too much detail about Types, for this is better suited to programming basic concept #3 – Data Structures. That's all I will touch on for now, but it will all make sense in time so don't worry!

In summation, we talked about what a **variable** is and how you can **store information** in a variable and then **retrieve that information** at some later point in time. The variable can have a **name** and this name you give to the variable is usually **named after the kind of content you'll be storing in the variable**, so if I'm storing your name in the variable, you'd name the variable "your Name." You wouldn't HAVE to give it that name. You could name the variable "Holy Crap I'm Programming," but that wouldn't make a

whole lot of sense considering you are trying to store a person's name. Makes sense right? Finally, variables have **types** and these types are used to help us organize what can and cannot be stored in the variable. *Hint: having a type will help to open up what kind of things we can do with the information inside the variable.* **Example:** if you have two Integers (let's say 50 and 32), you would be able to subtract one variable from the other (i.e. $50 - 32 = 18$). This is pretty straight forward right? But, if you had two variables that stored names (i.e. "Trevor" and "Geoff"), it wouldn't make sense to subtract one from the other (i.e. "Trevor" – "Geoff") because that just doesn't mean anything! **Types are also a powerful thing**, and they help us to make sense of **what we CAN do** with our variables and **what we CANNOT do!**

Control Structures



These are the 5 basic concepts of any programming language. Here's a breakdown again of those concepts:

1. *Variables*
2. ***Control Structures***
3. *Data Structures*
4. *Syntax*
5. *Tools*

We've already discussed what a variable is, so now let's talk about **control structures**. What on earth is a control structure!? Wiki describes it as follows:

A control structure is a block of programming that analyzes variables and chooses a direction in which to go based on given parameters. The term flow control details the direction the program takes (which way program control "flows"). Hence it is the basic decision-making process in computing; flow control determines how a computer will respond when given certain conditions and parameters.

That definition is obviously a bunch of technical terms that no beginner to programming would understand, so let me try to describe it in laymen terms. When a program is running, the code is being read by the computer line by line (from **top to bottom**, and for the most part **left to right**) just like you would read a book. This is known as the "**code flow**." As the code is being read from top to bottom, it may hit a point where it **needs to make a decision**. This decision could make the code jump to a completely different part of the program, or it could make it re-run a certain piece again or just plain skip a bunch of codes.

Think of this process like if you were to read or choose your own adventure book. For example, you get to page 4 of the book and it says "if you want to do X, turn to page 14, and if you want to do Y, turn to page 5. That **decision** that must be made by the reader is the same **decision** the computer **program must make**; however, only the computer program has a **strict set of rules to decide which direction to go** (whereas if you were reading a book, it would be a subjective choice based on whomever is reading the book). This decision that must be made will in turn **affect the flow of code**, which is known as a **control structure**!

Okay, that doesn't seem to be such a hard concept. A control structure is just a decision that the computer makes. However, what is it using to base that decision on? Well, it's simply basing its decision on the variables that you give it! Let me show you a simple example, here's a piece of Java code:

```
if (yourAge < 20 && yourAge > 12)
{
    // you are a teenager
}
else
{
    // you are NOT a teenager
}
```

You can see above that we have a variable and its name is `yourAge`, and we are comparing `yourAge` to 20 and 12. If you're less than 20 AND you're more than 12, then you must be a teenager (because you are between 13 and 19 years of age). What will happen inside of this control structure, is that if the value assigned to `yourAge` variable is between 13 and 19, then the code will do whatever is inside of the first segment (between those first two curly braces `{ }`), and it will skip whatever is inside of the second code segment (the second set of curly braces `{ }`). And if you are NOT a teenager, then it will skip the first segment of code and it will execute whatever is inside of the second segment of code.

Let's not worry too much about what the code looks like for the moment, as I'll touch on how to write the code out properly in section #4 **syntax**. The only concept you need to try and wrap your head around right now is that there is a way in programming to "choose" which lines of code to execute and which lines of code to skip. This will all **depend on the state of the variables** inside of your **control structure**. When I say **state of a variable**, I just mean what value that variable has at any given moment, so if `yourAge = 15`, then the state of that variable is currently 15 (therefore, you're a teenager).

You've now seen one control structure and I've tried to explain it as best I could. This control structure is known as an `if...else` structure. This is a very common control structure in programming; let me hit you with some other examples. Here's a `while` loop control structure:

```
while (yourAge < 18)
{
    // you are not an adult
    // so keep on growing up!
}
```

This `while` loop control structure is also very handy, and its purpose is to **execute code** between those curly braces `{ }` **over and over** and over until the **condition** becomes false. Okay, so what's the **condition**? Well, the condition is between the round brackets `()`, and in this example it

checks `yourAge` to see if you are less than 18. So if you are less than 18, it will continuously execute the code inside the curly braces `{ }`. Now, if you were 18 or older before the `while` loop control structure is reached by the code flow, then it won't execute ANY of the code inside of the curly braces `{ }`. It will just skip that code and continue on executing code below the `while` loop control structure.

There are a few other examples of control structures in Java, but I don't want to overwhelm you with them right now

We've learned that code flows from **top to bottom** and for the most part **left to right**, just like a book. We've learned that we can skip over certain codes or execute certain parts of a code over and over again, and this is all achieved by using different control structures. These control structures are **immensely important to programming**, for they make the programs function properly. For example, you wouldn't want people to be able to login to your Facebook account if they enter the wrong password right? Well, that's why we use the `if...else` control structure. If the passwords match, then you will be logged in or else a "password is incorrect" screen will be shown. So, **without control structures**, your program's code **will only flow in one way**, and it would essentially only do one thing over and over again, which wouldn't be very helpful. **Changing what the code does is based on a variable is what makes programs useful to us all!**



Data Structures

1. *Variables*
2. *Control Structures*
3. **Data Structures**
4. *Syntax*
5. *Tools*

Data structures, what are they, why are they useful? Well, let's turn to a quick definition from wiki:

In computer science, a data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.

Okay, that definition is a little more down to earth. But, there is a lot of 'meat' behind data structures. Let me try to explain the point of data structures by giving you an example... Let's use a list of contacts as the example! You probably have a list of contacts somewhere in your life, whether it's in one of your email programs or an address book in a kitchen drawer. There are a bunch of contacts, and that list of contacts could grow (or shrink) at any given moment. If you were to try and represent all those contacts as variables in a computer program, how would you do it? Well, there's a right way and a wrong way. For the purposes of our example, let's say we need to keep track of 10 contacts.

First, the **WRONG WAY**:

If we need to store 10 contacts, we would probably define 10 variables, right?

```
String contact1, contact2, contact3, contact4, contact5, contact6, contact7, contact8,  
contact9, contact10;  
contact1 = "John Smith (john.smith@someCompany.com)";  
contact2 = "Jane Smith (jane.smith@someCompany.com)";  
contact3 = //... etc.
```

In the world of programming, this is just a terrible way of trying to store 10 different variables. This is because of two main reasons:

1. The **sheer amount of text** that you'll need to write in your program
 - Sure, right now we only have 10 contacts, so it's not too bad, but what if we had 1,000 contacts! Imagine typing that out a thousand times! Forget about it!
2. The **flexibility** of code
 - If we need to **add another contact**, we wouldn't be able to do it without **manually editing our code**. We would have to go into our code, physically write out `contact11`, and then try to store whatever information is needed into the new variable. This is just crazy talk!

So, what is the **RIGHT WAY**?

I'm glad you asked, for it's a **data structure** of course!

In this case, we have a list of contacts, and with Java there is a data structure called a List! Okay, so what does it look like? Here's the code:

```
List contacts = new ArrayList();
```

Again, let's not worry about all those symbols and confusing brackets and semi-colon because we'll cover that later. All I need you to understand right now is that there is a way to **store a bunch of contacts** into something called a **data structure** (in the case of our example, a `List`). Okay, so, what's so great about a List? Well, for one thing, you can add and remove things from a list with ease. If you started with 10 contacts, it's a piece of cake to add another contact to the list. How you ask? Just like this:

```
contacts.add("John Smith (john.smith@someCompany.com)");
contacts.add("Jane Smith (jane.smith@someCompany.com);
```

Voila! We've added another contact to our contacts list! So, you may be saying, "The right way looks like just as much typing as the wrong way." You've got a point, but the main difference is that with the first approach, you had to "create" 10 unique variables (i.e. `contact1`, `contact2`, `contact3`, `contact4`...), but with the second approach, we only created 1 variable (`contacts`), for we only had to create 1 variable, and because we can say `contacts.Add(someRandomContact)` means that our code is much more flexible and **dynamic**. When I say **dynamic**, I mean that the **outcomes** of the program **can change depending on what variables** you give to it. That's really the key to using **data structures**! We want our code to be as **dynamic** as possible; we want it to be able to handle a bunch of situations without having to write more and more code as the days go by.

In essence, a **data structure** is just a **way to get around having to create tons of variables**. Now, there are a bunch of other data structures in Java, but I'll just quickly touch on the ones we use most often when programming. The next one I'll talk about is the `HashMap`. This doesn't sound as straightforward as our last data structure (the List), but it's really just a fancy name for a fairly simple concept. So, what is a hash map? Here's a visual representation:

"Honda" -> "Civic", "Prelude"
"Toyota" -> "Corolla", "Celica", "Rav4"
"Ford" -> "Focus", "Mustang"
"Audi" -> "R8"

What you see here is the **make** of a car **on the left**, which points to different **models of that make**. This is known in the programming world as a **Key/Value pair**. The **key** in this case is the **Make** of the car (i.e.

Toyota, Honda, Ford, and Audi) and the **value** in the case is the **model** (i.e. Civic, Corolla, Focus, and R8). That is how a `HashMap` works, and it's just another **data structure!** This concept is used all over the place on the web. For example, if you have ever filled out a quote for automobile insurance online, you've probably had to choose the make and model of your car. I bet you that information was presented on screen by using a `HashMap` (just like I showed you above)! When you select "Toyota" from one drop down list, the second drop down list will need to change its contents to only show models of cars made by Toyota. If you change the first drop down to Honda, the second drop down list will change again to show cars made by Honda.

How about I show you what a `HashMap` looks like in Java code?

```
Map<String, List<String>> cars = new HashMap<String, List<String>>();  
cars.put("Toyota," new ArrayList(Arrays.asList("Corolla," "Celica," "Rav4")));
```

Wow, there's a lot of stuff going on in there, but don't be intimidated because all you need to understand right now is the concept that this `HashMap` **data structure** stores **key/value pairs**. In our case, a **make** is the **key** and a `List` of models is our **value**. Now, that's kind of interesting. Did you see what we did there? I just said that we have a `HashMap` **data structure** and we are storing a `List` inside the `HashMap` (which is another **data structure**). This is done on a fairly regular basis in the programming world. We can **use one data structure** inside of **another!** So, if the point of data structures is to help us minimize the number of variables we need to create, then we are really saving ourselves the hassle of creating a lot of variables!

Okay, so let's sum up everything we've talked about. A **data structure** is a way of **storing** and **organizing data** in such a way that it can be **used efficiently**. The point is to avoid creating crap loads of variables people! Java allows us to do this by using `Lists` and `HashMaps` (as well as many other data structures, but these two are the most common). These two data structures can grow and shrink without you having to worry about coding all of that 'behind the scenes' stuff. When you add to a `List`, the new element is there and is usable (like magic). When you remove from the `List`, the element is now gone (poof!). This saves us from creating and deleting variables in our code manually.

Alright, that about sums up what I need you to know for our next section: **syntax**. In my opinion as a veteran programmer, **syntax** is the one thing that will discourage you from learning to program. It takes practice and patience to understand, but I promise you, it will become second nature in time. So please be patient and be excited to learn about **syntax** in Java because once you get the hang of it, you'll be on your way to programming like a champ!



Syntax

Welcome back to our fourth lesson in our five part series on the 5 basic concepts of any programming language.

1. *Variables*
2. *Control Structures*
3. *Data Structures*
- 4. Syntax**
5. *Tools*

What is syntax? As always, let's hop over to wiki for a quick definition:

In computer science, the syntax of a programming language is the set of rules that define the combinations of symbols that are considered to be correctly structured programs in that language.

Alright, so I would say that's almost English, but what do they mean by "combinations of symbols that are correctly structured?" Well, I would choose a different word than symbols. I would define **syntax** to be a particular **layout of words and symbols**. An example of this in the case of Java would be round brackets (), curly brackets {}, and variables, among other things. Think of it like this: when you look at an **email address** (i.e. `john.smith@company.com`), you can **immediately identify** the fact that it's an **email address** right? So **why is that?** Why does your brain make the connection that it's an email address and not a website address? Well, it's because an **email address has a particular syntax**. You need some **combination of letters and numbers**, potentially with underscores (_) or periods(.) in between **followed by** an at (@) **symbol** and by a **website domain** (company.com). That is a defined combination of letters and symbols that are considered correct structure in the "language" of the internet and email addresses.

So, **syntax** in a programming language is much the same, for there are a **set of rules** that are **in place**, which when you follow them, **allows** your **programming language** to **understand** you and allow you to create some piece of functioning software. However, if you don't abide by the rules of a programming languages' syntax, you'll get errors 😞

How about an example of **syntax in Java**? Well you've seen it already back when we talked about variables and control structures. To **define a variable** in Java, you need to do this:

```
String helloVariable = "Hello Everyone!";
```

There are **four parts** to the syntax of creating a variable in Java. The **first is** the word `String`. This is the variable's **type**. Remember when we talked about variable **types** in the first part of this series? I mentioned `String`, `Integer` and `Double`. These are three different variable **types** that allow you to

store three different **kinds of data**. A `String` in this case, allows you to store regular letters and special characters.

The **second part** to this variable creation syntax is the **variable name**, and in this case I arbitrarily chose `helloVariable`. I could have just as easily chosen `holyCowThisIsAVariableName`. Variable names can be made up of letters and numbers, but the only special characters they can contain are underscores (_). They also usually start with a lower case letter, but they don't have to. That's just kind of an accepted and suggested convention (at least in the Java world). The **third part** of the syntax for creating a variable is the **value that the variable will hold**. In this case, we have a `String` variable, so we have the value "Hello Everyone!"

In java, Strings are defined by wrapping regular letters/numbers/special characters in quotes (""). Again, that's just the **syntax** that Java uses. The **last part** of this **syntax** is the part that **marks** this particular code segment **as being complete**. In Java, we use the semi-colon (;) to mark a part of our code as complete. You will see that almost every line of code in Java will end with a semi-colon (;). There are certain exceptions to this; for example, control structures aren't marked with semi-colons but they use curly braces to make their beginning and end. Think of it like putting a period at the end of every sentence. If we didn't put a period, we would just have one long unstructured run on sentence, and that wouldn't help us to understand anything that's being said.

So, as I mentioned before, the **syntax** of any programming language will **likely** be your **biggest hurdle as a new developer**, but as you see more and more examples of code and are introduced to more and more **syntax** in the language, you will become comfortable. **There is good news** though, for people have realized that dealing with syntax can be tough, so certain companies (or groups of enthusiasts, a.k.a nerds) have created **tools** to help us with the **syntax** of programming languages. These **tools** are called **IDEs or Integrated Development Environments**, which you can download onto your computer and use to create programs. These **IDEs** have built in **syntax checkers** (much like the grammar checker in MS Word) that will let you know if your **syntax is incorrect**, and it will even give you hints with what it thinks you meant to put! Don't you worry because I'll cover those tools in the next section of this chapter.

Let's sum up. We have learned that **syntax** just means that there's a "**correct**" way to **write down your code**, and this allows the **programming language understand what** it is that **you're trying to tell it** to do. Unfortunately for us, computers can't read our minds (yet!) and know what it is that we want them to do, so some very smart people have created this "computer language;" when understood by programmers, it allows us to tell the computer what actions we would like it to carry out and whether that action be to send a bill payment to our credit card company or to play a game of poker online with a virtual table full of strangers. Syntax is our systematic way to talk to a computer and convey our wishes.

I hope I have taken a little bit of mystery out of the term syntax, and I look forward to teaching you about our final subject... **tools!** A developer's best friend 😊

Tools

Welcome back to our fifth lesson in our five part series on the 5 basic concepts of any programming language. In this java tutorial, the concept we'll talk about concept is **tools**.

1. *Variables*
2. *Control Structures*
3. *Data Structures*
4. *Syntax*
5. **Tools**

What are tools? Well, I don't think we need to go to wiki to define this one, as many of you should already know what a tool is. In the real world, a tool is something (usually a physical object) that allows you to get a certain job done in a timelier manner. Well, this holds true within the programming world too. A tool in programming is a piece of software that, when used while you code, allows you to get your program done faster!

There are probably tens of thousands, if not hundreds of thousands of different tools across all the programming languages, but I'll focus on the main tools that everyone is likely to use.

The first and most important tool, in my opinion, is an IDE. An Integrated Development Environment is a piece of software that will make your coding life so much easier. IDEs will check the syntax of your code to ensure you don't have any errors, they will organize your files and give you a nice way to view them (i.e. applies color schemes to your code so it's easier to interpret), they tend to have code completion (which will actually fill in some code for you, in common scenarios), and allow you to navigate through your code easily. There are many other advantages of using an IDE, but I think you get the idea. In the Java world, the IDE I use most often is:

- [Spring \(STS\)](#)

This IDE is free and full of features, it's what I use whenever I program (both at work and at home)

For the purposes of this java tutorial, I will focus on just this one tool, as it will be the most important thing you use when you begin programming. So, let's install Spring STS! The first thing you'll need to know is that the Spring STS IDE, and all IDEs in Java require the Java Development Kit to be installed on your computer, so how about we [download that now as well here](#).

Step 1 – To download the Java Development Kit (JDK), go to the [Oracle Java Development website](#)

Java SE
Java EE
Java ME
Java SE Support
Java SE Advanced & Suite
Java Embedded
JavaFX
Java DB
Web Tier
Java Card
Java TV
New to Java
Community
Java Magazine
Java Advanced

Java SE Downloads

Thank you for downloading this release of the Java™ Platform, Standard Edition Development Kit (JDK™). The JDK is a development environment for building applications, applets, and components using the Java programming language.

The JDK includes tools useful for developing and testing programs written in the Java programming language and running on the Java™ platform.

Looking for the JavaFX SDK?
The JavaFX SDK is available [here](#).

Java SE Development Kit 6 Update 33

You must accept the [Oracle Binary Code License Agreement for Java SE](#) to download this software.

Accept License Agreement Decline License Agreement

Product	File Description	File Size	Download
Linux x86		65.42 MB	jdk-6u33-linux-i586-rpm.bin
Linux x86		68.42 MB	jdk-6u33-linux-i586_bin
Linux x64		65.64 MB	jdk-6u33-linux-x64-rpm.bin
Linux x64		68.69 MB	jdk-6u33-linux-x64_bin
Solaris x86		68.33 MB	jdk-6u33-solaris-i586.sh
Solaris x86		119.88 MB	jdk-6u33-solaris-i586.tar.Z
Solaris SPARC		73.3 MB	jdk-6u33-solaris-sparc.sh
Solaris SPARC		124.45 MB	jdk-6u33-solaris-sparc.tar.Z
Solaris SPARC 64-bit		12.18 MB	jdk-6u33-solaris-sparcv9.sh
Solaris SPARC 64-bit		15.59 MB	jdk-6u33-solaris-sparcv9.tar.Z
Solaris x64		8.44 MB	jdk-6u33-solaris-x64.sh
Solaris x64		12.24 MB	jdk-6u33-solaris-x64.tar.Z
Windows x86		69.66 MB	jdk-6u33-windows-i586.exe
Windows x64		59.67 MB	jdk-6u33-windows-x64.exe
Linux Intel Itanium		53.94 MB	jdk-6u33-linux-ia64-rpm.bin
Linux Intel Itanium		60.65 MB	jdk-6u33-linux-ia64_bin
Windows Intel Itanium		57.86 MB	jdk-6u33-windows-ia64.exe

Java SDKs and Tools

- [Java SE](#)
- [Java EE and Glassfish](#)
- [Java ME](#)
- [JavaFX](#)
- [Java Card](#)
- [NetBeans IDE](#)

Java Resources

- [New to Java?](#)
- [APIs](#)
- [Code Samples & Apps](#)
- [Developer Training](#)
- [Documentation](#)
- [Java.com](#)
- [Java.net](#)
- [Student Developers](#)
- [Tutorials](#)



Step 2 – Run the downloaded file and leave all options default as you click next

Special Note: This installer will likely **popup another installer** for the Java Runtime Environment (JRE), so if it looks like your install has stopped for no reason, check to see if you have any other pop-ups that may be hidden that require you to click **Next**.

Step 3 – Click [this link](#) to go to the Spring STS download page and skip the registration (if you like):

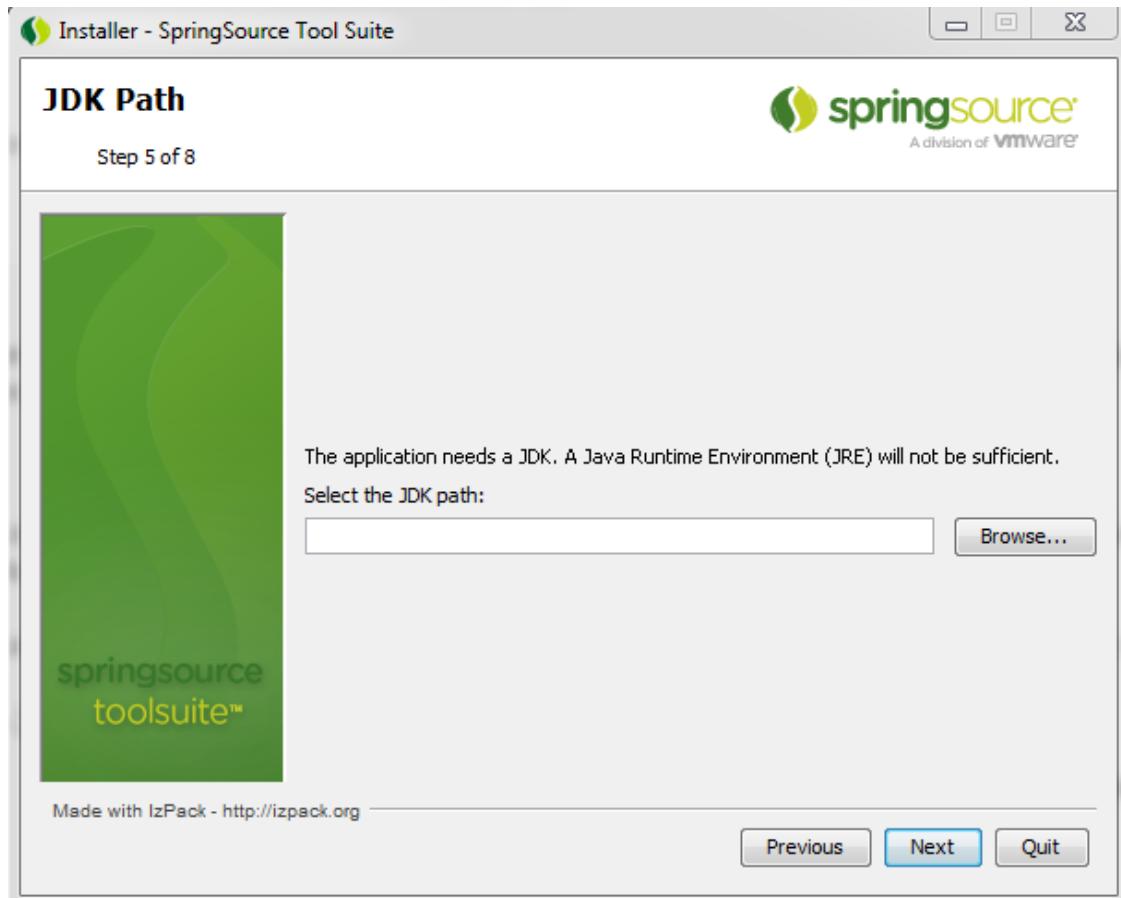
The screenshot shows the SpringSource Tool Suite Download page. At the top, there is a registration form with fields for First Name, Last Name, Company, Role (a dropdown menu), Email, Phone, and Zip/Postal code. Below the form is a checkbox for accepting license terms and export restrictions, followed by a link to 'take me to the download page'. A blue arrow points from this link towards the bottom of the page. To the right of the form, there is a section titled 'UPCOMING EVENTS' featuring a banner for 'springone 2012' in Washington DC. Below the banner, there are links for 'JULY 12, 2012 What's new in RabbitMQ?' and 'JULY 26, 2012 Webinar: Introduction to Spring Data Neo4j'.

Step 4 – Choose the appropriate version of STS to install. I've highlighted the windows versions, but I believe there is a MAC version as well:

The screenshot shows the SpringSource Tool Suite Download page. It includes sections for 'GETTING STARTED' (with links to 'New and Noteworthy', 'Feature Comparison', and 'Installation Instructions') and 'DOWNLOADS' (with a note about native installers). The main focus is a table listing the 'CURRENT VERSION - 2.9.2.RELEASE' of STS. The table has columns for 'Description', 'Link', 'Size', and 'Hash'. Four rows are shown, each corresponding to a different Windows version: 'Windows' (link: springsource-tool-suite-2.9.2.RELEASE-e3.7.2-win32-installer.exe), 'Windows' (link: springsource-tool-suite-2.9.2.RELEASE-e3.7.2-win32.zip), 'Windows (64-bit)' (link: springsource-tool-suite-2.9.2.RELEASE-e3.7.2-win32-x86_64-installer.exe), and 'Windows (64bit)' (link: springsource-tool-suite-2.9.2.RELEASE-e3.7.2-win32-x86_64.zip). Blue arrows point from the highlighted 'Windows' links in the first two rows towards the bottom of the page.

Description	Link	Size	Hash
Eclipse 3.7.2	springsource-tool-suite-2.9.2.RELEASE-e3.7.2-win32-installer.exe	352MB	sha1 - md5
Windows	springsource-tool-suite-2.9.2.RELEASE-e3.7.2-win32.zip	351MB	sha1 - md5
Windows (64-bit)	springsource-tool-suite-2.9.2.RELEASE-e3.7.2-win32-x86_64-installer.exe	352MB	sha1 - md5
Windows (64bit)	springsource-tool-suite-2.9.2.RELEASE-e3.7.2-win32-x86_64.zip	351MB	sha1 - md5

Step 5 – Now, once you've downloaded that file, run it and start the installation process. Leave all the options set to their defaults as you click next. Do it until you get to this screen.



Here you will choose the directory where you installed your Java SDK. The default location is:
'C:\Program Files\Java\jdk1.6.0_33'

There, now you have successfully installed Spring STS and you are ready to begin programming! That's all I will cover. You are now ready to learn how to write your first java program (also known as the Java Hello World program).

Chapter 2

Your First Java Program

The Java Hello World Program

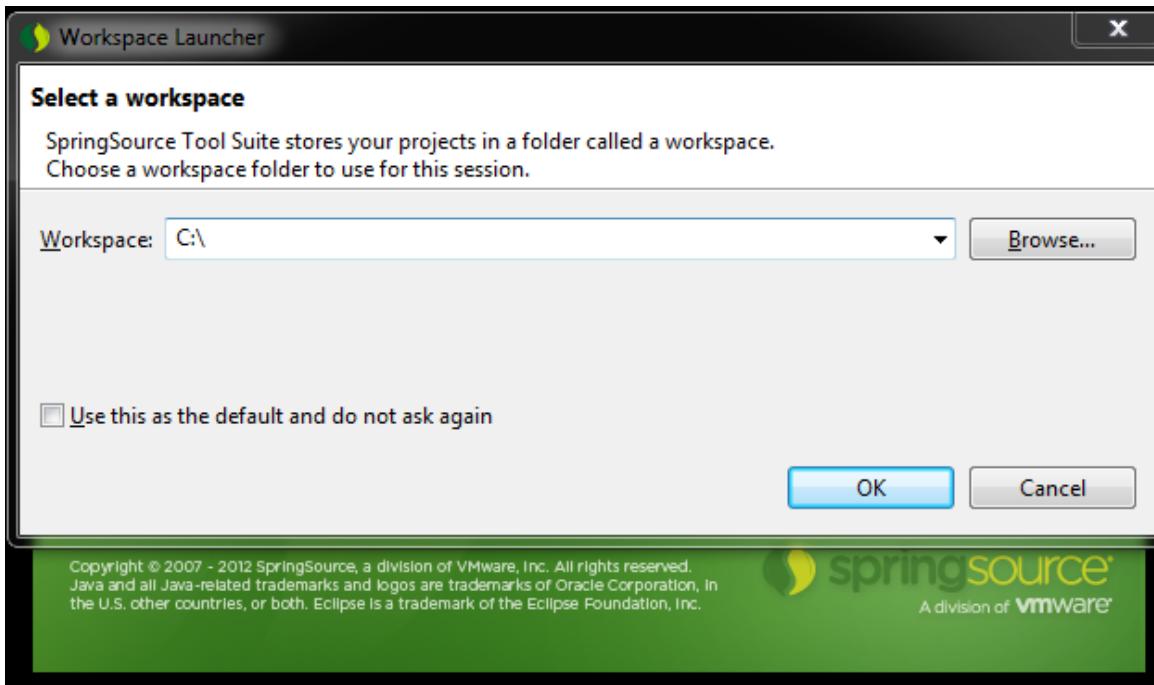


So, what does Java Hello World even mean anyway? Well, in the programming world, the term “Hello World” typically refers to the most basic program that can be written and run in a given language. Think of it like this, if you can write a Java Hello World program, this means that you were able to:

1. *Setup the necessary tools*
2. *Write the code*
3. *Correctly compile the code*
4. *Run your program*

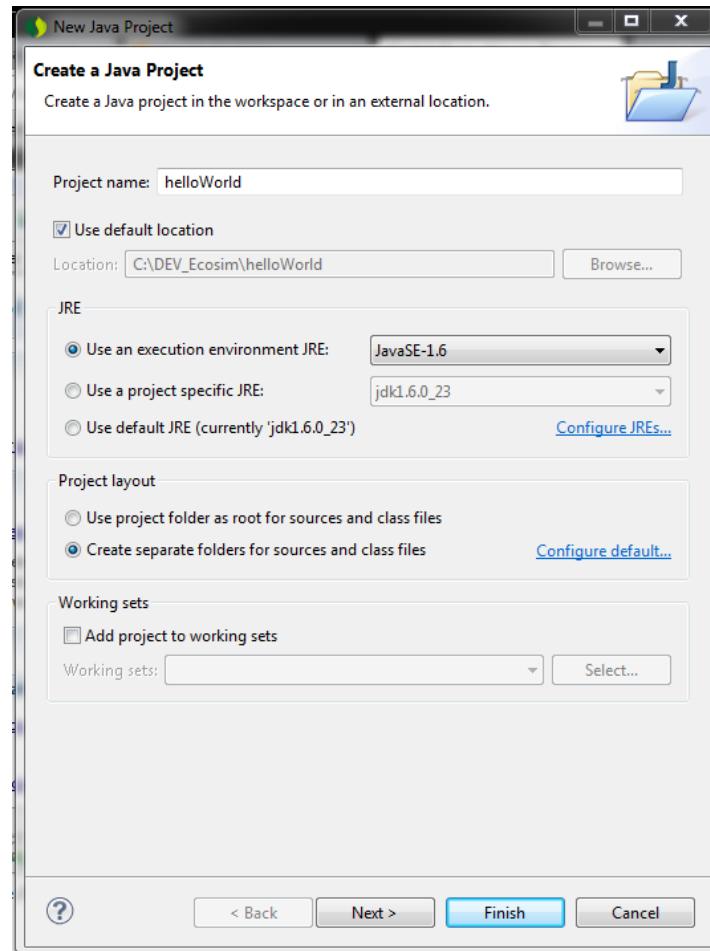
This isn’t always the easiest task to accomplish when you are new to a programming language. So, as it pertains to Java, if you can accomplish all of these tasks and successfully write your Java Hello World program, then you are off to a good start. Now, we have already covered step 1 above, in Chapter 1. So let’s move on to step 2, writing the code.

First, launch your Spring STS tool, you’ll notice that you get stuck on this screen:



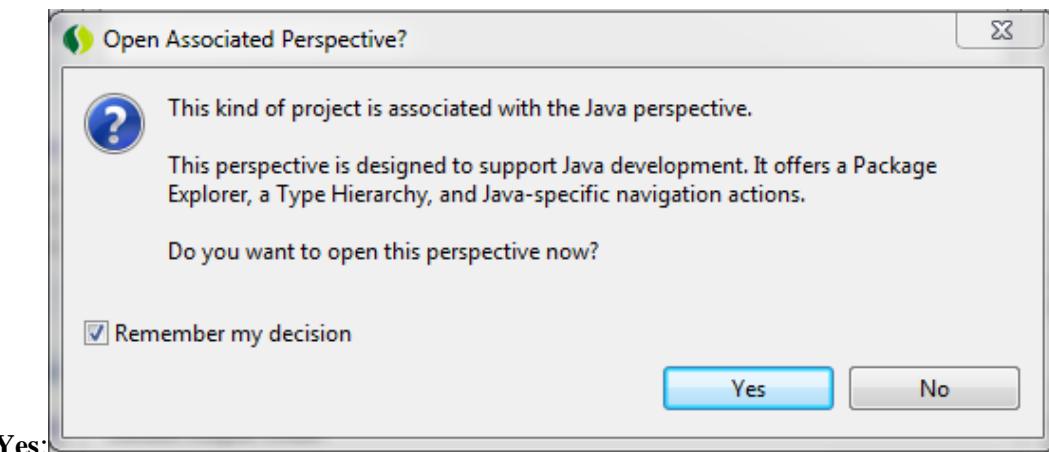
You’ll just need to pick a folder that will house your development files. I usually choose C:\DEV\, but it’s your choice. So, pick a folder and click OK.

Now, with Spring STS open, you'll need to start a new project, so choose File -> New -> Java Project. You'll be presented with this screen:



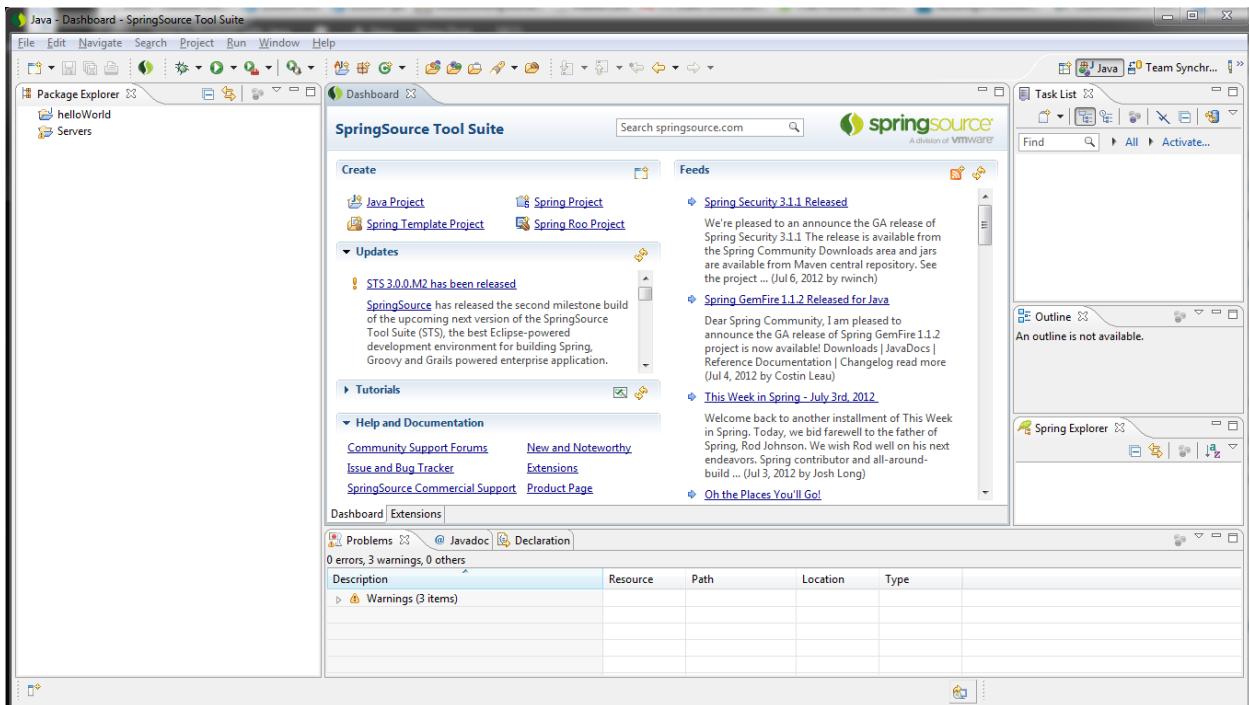
Be sure to fill out a program name. I used “hello World,” but you can use any name you like for this project. Next, you should have the JRE installed so **Click Finish**.

You may or may not be presented with this window, but if you are just click the “Remember my decision” and **click**

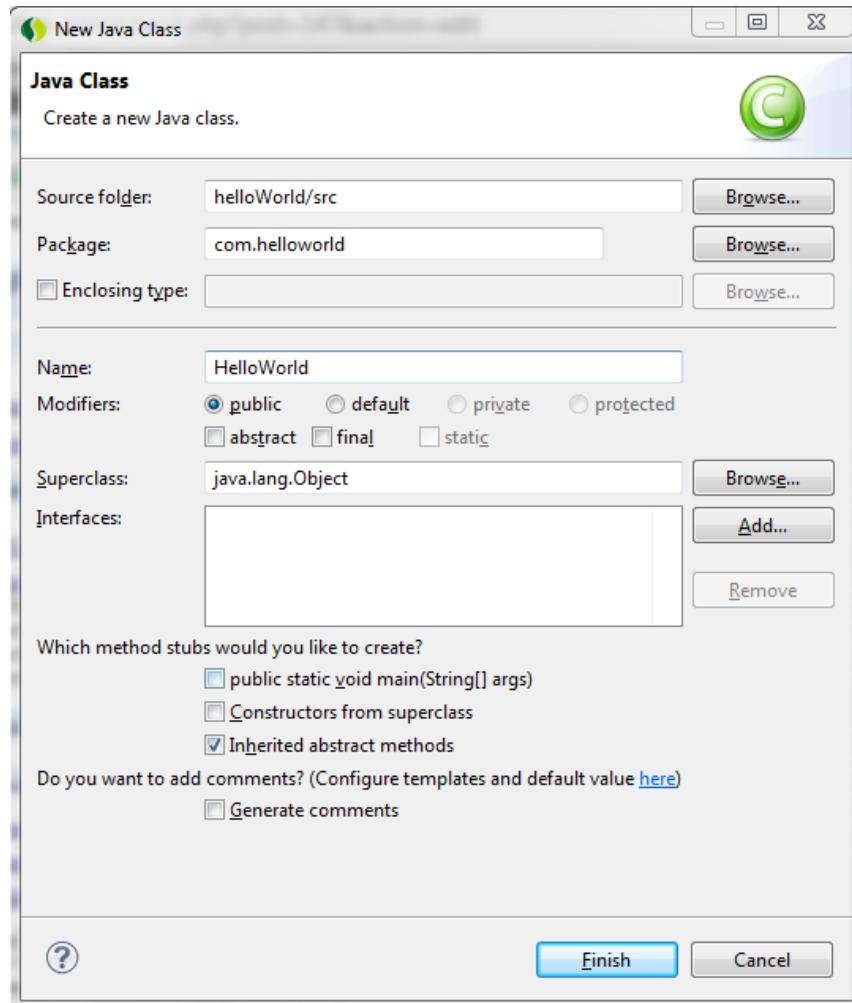


Yes:

Now, if all went well, you should see something like the following:

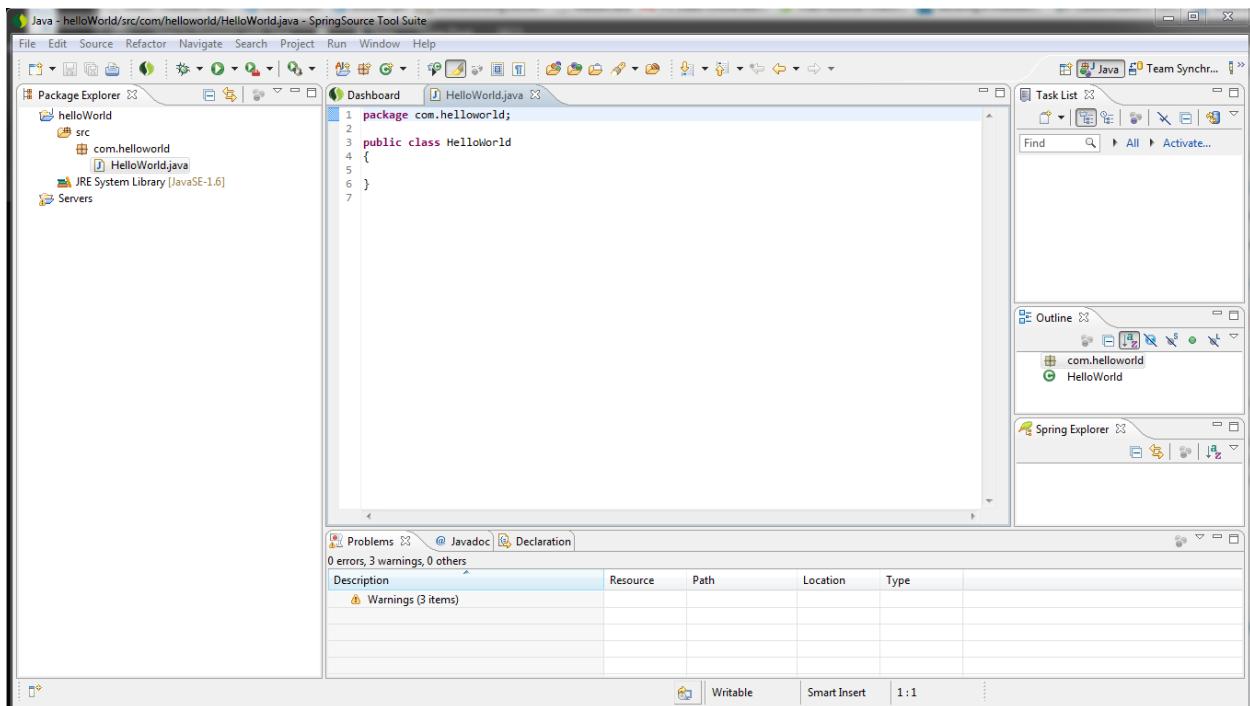


We need to create a new file for this project, so click File -> New -> Class. You should now see this screen:



Be sure to fill out two fields: **Package** and **Name**. The Package I used was "com.helloworld." and the Name I used was "Hello World." You can see this in the screenshot above. Now click **Finish**.

This should have created a new Class file called `HelloWorld.java`, and you should now see something like this on your screen:

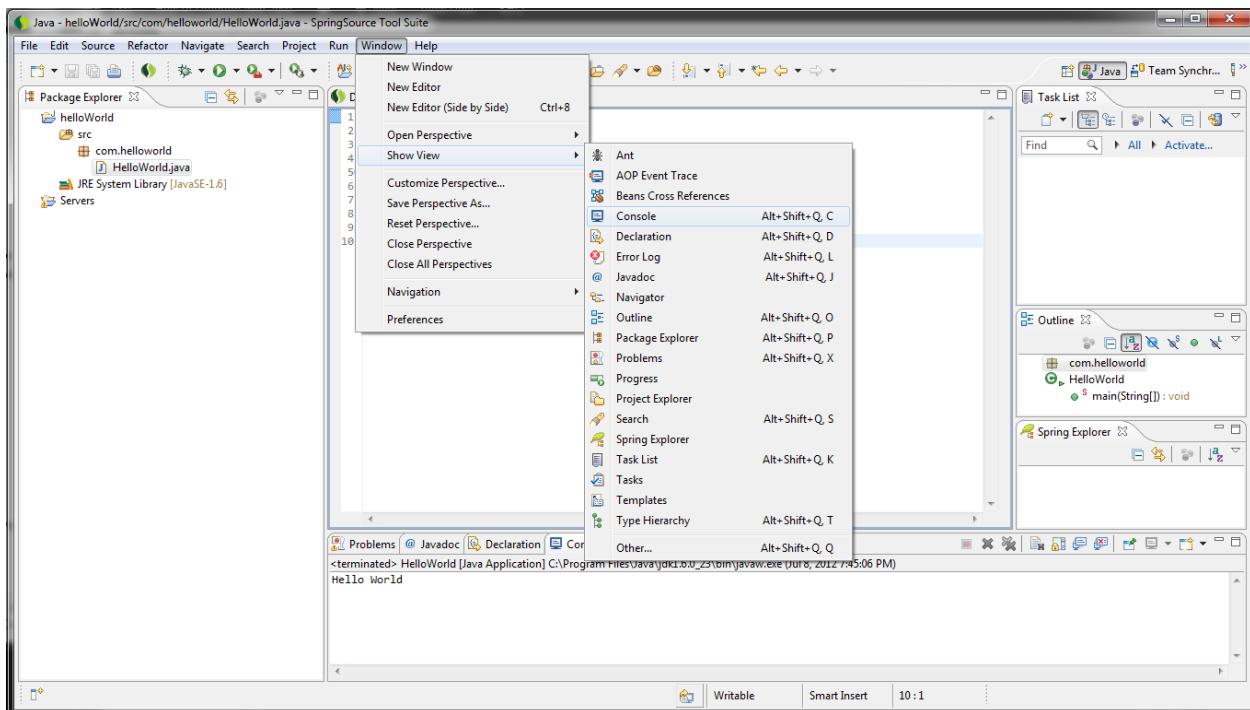


Now we are ready to write some code! If you like, you can copy/paste the following code into your `HelloWorld.java` file:

```
package com.helloworld;

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

Now, for the final step, you'll need to run this! If you right click, your mouse anywhere in the code area, you'll get a menu that will have a menu item named “Run As,” hover over it and select “Java Application.” This will run your code and output “Hello World” into your console. If you don't see the **console**, just click Window -> Show View -> Console, like so:



Voila, you have just setup, coded, and ran your first Java program! Welcome to your first accomplishment as a Java developer.

Java Hello World (part II)

Now that we've created our first Java program, we need to figure out what all that code actually means right!? Surely you're looking at it and saying, "How does anyone understand that stuff?" I assure you it's not as scary as it looks. All we need to do is break down each part of the code and understand what each part does and why it's important, so let's do that! Here's the code:

```
package com.helloworld;

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

For now, let's skip over the first line:

```
package com.helloworld;
```

It was created automatically by our IDE (the Spring STS program you're using to code) when we created the `HelloWorld.java` file. We'll talk about why this line is important later, but for now just know that it needs to be there or we'll get errors.

The second line is:

```
public class HelloWorld
```

This line is important for two reasons. The first reason is that it represents the Class. Now when I say Class, I don't mean a class of students with a teacher. I mean a Java Class file. These Class files represent a 'blueprint' for an Object in Java. Objects, in Java, are the foundation of the programming language, because Java is an Object Oriented programming language. Objects in Java represent exactly what they sound like, an Object in real life. Think of an Object like a noun (person, place or thing). Our example of `HelloWorld.java` isn't a great one, as Hello World doesn't really mean anything in the real world.

So, the second reason that this line is important is that the name "Hello World" has to be exactly that name because that's the name of the file that we created (`HelloWorld.java`). We chose to name this file

“HelloWorld.java” for no particular reason, but since we chose that name, the name of our declared Class needs to match. One thing to note is that Java is case sensitive, so that means “Hello World” is not the same as “hello world.” Capital and non-capital letters matter.

One thing I haven’t yet touched on with this line of code is the first word, “public.” In Java, there are four levels of visibility for your code: public, protected, package and private. Essentially all this means is whether or not other Class files will be able to have access to this particular file. When we say public that means that any other file can look and interact with this Class file. This concept of visibility is also used with something called methods. What’s a method you ask? Good segue into the next line of code:

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

The first thing you may notice is that word “public” again. It means the same thing as it did before, except now it’s saying that this particular chunk of code (the code between the curly braces {}) is also public. Again, this means that any other Class files can access this particular chunk of code! This seems like a pretty easy concept right?

Now, I said that I would segue into what a method was. Well, you see how I keep referring to this code as a “chunk” of code. Well, now it’s time for me to use the technical term for this chunk of code. It’s called a method! Methods are essentially “chunks” of code that you can run over and over again by “calling” that “chunk” of code. “Calling a method” means that I wish to execute all the lines of code that are present between those curly braces {} of that method. Furthermore, we always give names to our methods just like we always give names to our variables. For this particular piece of code, the method’s name is “main.”

So now allow me to re-iterate:

```
public class HelloWorld
{
```

This is the Class definition.

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

This is a method within our Class definition.

We establish this concept of something being “within” our Class definition by looking at the curly braces {}. The outer set of curly braces is the Class definition, the inner set is the method definition. This is more or less constant in any Class file, so keep that in mind.

The two words on the left of this method “static void” are what are known as modifiers, for they modify how the method will work. Let’s ignore them for now because I will expand on what they mean in a

future java tutorial. That just leaves us with `(String[] args)`. This is known as the “parameter” or “argument” section of the method. You see, in Java’s syntax, a method can have variables passed in so we can use them and modify them inside the method if we wish. This is denoted with the parenthesis () after the method name.

The last thing I want to quickly point out is the `String[] args`. I’ve mentioned what a `String` was in our first lesson on variables, but why are there square brackets [] after the `String` variable type? This is just Java’s syntax for an Array. I mentioned what an Array was in the Data Structures lesson, but to refresh your memory, just think of it as a list of `Strings`. Think of it like a kind of grocery list: “Lettuce,” “Tomato,” “Bananas,” “Pasta sauce,” etc. This could represent an Array of `Strings` and this Array would be referenced by the variable name “args”! See?

`String[] args`

Now, our final line of code to analyze:

```
System.out.println("Hello World");
```

This one is pretty simple to explain. All this code does is output whatever is in quotes into your Console window in your IDE (Spring STS). This line of code is used most often when you just want to quickly take a look at the state of a variable at any given moment. For example, if you had a variable called `outsideTemperature` and this variable just updated its own value every 30 minutes. Let’s say you want to see what the variable’s state was (in other words, see what the outside temperature is); then you could write `System.out.println(outsideTemperature)`. The value that the variable has would then display in your Console window in your IDE.

Chapter 3

The Basics



What is a Method?

What is a method in Java? Well, we've already touched on what a method is in the [Java hello world](#) section, but I want to get more **return** in depth on the power of methods (or functions). So first off, you've never heard me say the word "function" before, so **before I talk about methods**, allow me to start with an **explanation of what a function is**. In Java, **methods and functions are really the same thing**. For the sake of argument, I'll just use the term method from now on. For example, a method in Java could look like this:

```
public Integer addTheseTwoNumbers1 (Integer firstNumber, Integer secondNumber)
{
    return firstNumber + secondNumber;
}
```

Now, this method is a little different. There are two main things that are different. The first thing is the code `public Integer addTheseTwoNumbers1`, and you see how we've put the word `Integer` next to the `public` keyword. This indicates that **this block of code will return** (or "spit out") an `Integer` value. The second thing that indicates that this block of code is different is the fact that it **has a return statement**. When you specify, you're saying that this block of code will be returning (or "sending back") whatever is to the right of it. How about we **see what that same block of code would look like if it was a method that didn't return anything**:

```
public void addTheseTwoNumbers2 (Integer firstNumber, Integer secondNumber)
{
    Integer aNumber = firstNumber + secondNumber;
}
```

Can you **spot the differences**? First, there's no code saying `public Integer`, but it says `public void`. The `void` modifier **indicates that this block of code won't be returning anything**. The second thing to notice is that there's no more `return` keyword in the method.

So, what is a method in Java? Well, a **method** is that a piece of code that **could perform some operations, then return** something back to whatever happened to "call" the method. Furthermore, a **method** will perform some operations, without needing to **return anything when it's done**, and the flow of code will just continue back from whatever called the method. Either way, a method will execute some block of useful code that can be called repeatedly from anywhere in your program (as long as it's declared `public`).

To illustrate how the code flow could differ between method calls, let's take a look at this program:

```
public class MyProgram
{
    public static void main(String[] args)
    {
        //-----//
        // here's our method call that returns a value //
        Integer resultOfAddition = addTheseTwoNumbers1(5, 24);
        //-----//
        System.out.println(resultOfAddition);

        //-----//
        // here's our method call that doesn't return a value //
        addTheseTwoNumbers2(3, 13);
        //-----//
    }

    public static Integer addTheseTwoNumbers1 (Integer firstNumber,
                                              Integer secondNumber)
    {
        return firstNumber + secondNumber;
    }

    public static void addTheseTwoNumbers2 (Integer firstNumber,
                                              Integer secondNumber)
    {
        Integer addedValue = firstNumber + secondNumber;
    }
}
```

If you were to copy/paste this code into your IDE (Spring STS) and run it, you'll see the following output:

29

This may or may not surprise you. If I were to look at this for the first time, I'd probably guess that you would see 29 and then 16. I would guess this because I see `addTheseTwoNumbers1(5, 24)` and `addTheseTwoNumbers2(3, 13)`, so naturally $5 + 24 = 29$ and $3 + 13 = 16$. Why didn't we see these two numbers as the output? Well that's because for one of the calls we used a return statement and for the other we didn't. Since one method **returns a value**, we can then use that value and display it in our console (**by using** `System.out.println()`). Whereas with the second method, we **don't** have a value being returned, so all that happens is that the two numbers are added together, and then we **exit the method** and the code continues to flow **without invoking** the `System.out.println()` code.

Why don't we just always use methods that return values? They seem more useful because they give us something that we can work with. Well, that's just because a **method that returned a value was more useful** to use **with this particular example**. There are times when you don't need something returned; for example, if you created a method to send an email to someone:

```
public void sendEmail(String contactAddress)
{
    String emailContents = "this is a test email.";
    Email.send(contactAddress, emailContents);
```

```
}
```

You see? We don't want anything back, but we want to have the code send the email and go about its business. Okay, so I hope I've helped answer the question "What is a method in Java?" and that you understand everything entirely.

Object Oriented Programming in Java

Java is known as an Object Oriented language. **What does Object Oriented mean?** It means that the **foundations** of any kind of **program constructed in Java** might be imagined in terms of **Objects**. A good example of this idea should be to have a look at a handful of sample business requirements for a product. Imagine that we are tasked with constructing a software program intended to keep track of an actual **public library system**. This system must keep track of each of the **branches** associated with the libraries, the whole set of **materials** that can be contained in the branches, and all of the **people** that may need to access **books** from the library's branch.

The first thing we're able to do is examine those specs and **pinpoint all of the keywords which happen to be nouns**. For the record, a noun is actually a person, place or thing. This means, if we analyze our requirements, we discern these particular nouns:

- 1) Library
- 2) Book
- 3) Branch
- 4) Customer

All these words represent Objects in Java. This really is, in essence, Object Oriented programming (generally known as O-O programming). What we can now go about doing, is in fact arrange these four Objects on to some sort of piece of paper and begin to distinguish what sort of attributes each one of these Objects contains. What do I mean by the attributes? Well, in O-O development, this is known as identifying the "has a" relationships. To provide an example, a `Branch` "has an" address, a `Book` "has a" title, a `Customer` "has a" name. We will **map out the most important attributes** that each one of these Objects contain and build ourselves a terrific starting point for the **design** of our Java application.

Object Oriented development allows developers to **think in terms of real life "things"** or **Objects** and simply solve issues with all those Objects. It's important to remember that Java is actually not the only O-O programming language in existence, as it was initially started nearly five decades ago and plenty of modern programming languages utilize Object Oriented principles. Some of these languages include C++, C#, Objective-C, Python, Ruby, and Visual Basic.

What would the Objects from our example look like in code? Let's take a look at the `Book` Object:

```
public class Book
{
    private String author;
    private String title;
    private Integer isbn;
    private Integer numberOfPages;

    public String getAuthor()
    {
        return author;
    }
    public void setAuthor(String author)
    {
        this.author = author;
    }
}
```

```

    }
    public String getTitle()
    {
        return title;
    }
    public void setTitle(String title)
    {
        this.title = title;
    }
    public Integer getIsbn()
    {
        return isbn;
    }
    public void setIsbn(Integer isbn)
    {
        this.isbn = isbn;
    }
    public Integer getNumberOfPages()
    {
        return numberOfPages;
    }
    public void setNumberOfPages(Integer numberOfPages)
    {
        this.numberOfPages = numberOfPages;
    }
}

```

Okay, so what can we identify in this code that we are already familiar with? Well, we certainly see the word `public` a lot sprinkled around here. To refresh your memory, the word `public` is a **modifier** that allows for any **Java Class** to have **access** to the **code within** the **scope** of what it's modifying. So, you see that the first `public` keyword is placed on the Class name, which in this case is `Book`.

This means that our `Book` Class **will be accessible by all other Classes in our project**.

Note: You may have noticed I used the terms Class and Object to describe the same thing. In actual fact, they are very similar, but the only difference is that the Class can be considered the blueprint for an Object. An Object is what is physically created when the Java program is running. So if I want to "create" a `Book`, let's say a "Harry Potter" `Book`, I would "instantiate" (or "create") a `Book` Object based on its Class blueprint. Make sense?

Let's move on to the `private` modifier that you see on the first four variables (or **attributes**) of our `Book` Class. This is kind of strange you might think. If a `Book` has attributes like a title, an author etc.; then **why are these marked as private**? Wouldn't we want other Classes to have access to a `Book` title for instance? Yes and no, for this approach used here is called **encapsulation**, and it's one of the **fundamental principles in Object Oriented programming**. Let's flip over to Wiki for a definition of encapsulation:

In a programming language, encapsulation is used to refer to one of two related but distinct notions, and sometimes to the combination thereof:

- A language mechanism for restricting access to some of the object's components.
- A language construct that facilitates the bundling of data with the methods (or other functions) operating on that data.

In our example, we are touching on the **first part of this definition**, which is **restricting access**. We don't want just anyone to be able to come in and say, change the title of our `Book` Object. We want to be able to "screen" or "moderate" what changes are allowed and which are not. So we accomplish this by setting the attributes of our `Book` to be `private` and introduce `public` methods where you'll be able to retrieve and/or change the value of our `Book`'s attributes. So, if you want to know what a `Book`'s particular title is, you would have to "ask" like this:

```
String aBooksTitle = book.getTitle();
```

And just the same, if you wanted to change a particular `Book`'s title, you would do this:

```
book.setTitle("Harry Potter Vol. 2");
```

If the reasoning behind why we are doing this doesn't seem to make sense at the moment, **don't worry**. This approach is used over and over in Object Oriented programming languages, so you will become familiar with it and it will make sense in time. The main take-away for this Java tutorial is to know that **Object Oriented programming just means that we can create programs in terms of real life Objects**, and those Objects have **ways of interacting with each other!** This is a piece of cake right?

Nested IF Statements

Here's a quick tutorial on the IF statement and some of its more complex uses.

The nested IF

At this point, you are all fairly comfortable with the `if` statement as it's the most basic control structure in the Java programming language (and other languages too), but were you aware that you could nest these `if` statements inside of each other?

How about I first introduce a simple example to refresh your memory!

```
int age = 29;

if (age < 19)
{
    System.out.println("You are not an adult.");
}
else
{
    System.out.println("You are an adult.");
}
```

As you can see here we have a standard `if..else` structure. In this particular example I've set the `age` variable equal to the `int` value 29. So, since you are very keen with your Java knowledge, I bet you'd guess that the console output would read "You are an adult." Nice! So let's get a little more complicated shall we?

```
int age = 29;

if (age < 13)
{
    System.out.println("You are but a wee child!");
}
else if (age < 19)
{
    System.out.println("You are no longer a child, but a budding teenager.");
}
else
{
    if (age < 65)
    {
        System.out.println("You are an adult!");
    }
    else
    {
        System.out.println("You are now a senior, enjoy the good life friends!");
    }
    System.out.println("Also, since you are over the age of 19, you deserve a drink!");
}
```

```
}
```

Alright! So, you see here how I made use of a nested `if` statement? Since my `age` variable is still set as 29, the code will flow to the first `if` and check to see if the `age` is less than 13. We all know that 29 is **not** less than 13, so we continue downward to the first `else if`. Now We check if `age` is less than 19, it's not, so we continue to the next `else if`. But here's where it gets interesting, there are no more `else if` statements, all we have left is an `else` block. So, what happens?

Well, the same rules always apply to the flow of code in an `if..else` condition. If the none of the conditions evaluate to `true` in all the `if` conditions, then the code will automatically choose the `else` path if it exists. In our example, it **does indeed exist**. So the code will choose that path. And what do you know, we've now encountered `..else` block of code. So, as predictable as the sun will rise and set, Java will follow the same set of rules for this new `if..else` block.

We will now evaluate if `age` is less than 65. And as luck would have it, our variable is set to 29. So, yes, 29 is less than 65, so we'll execute the code inside of this block and the console will output "You are an adult."

Now, here's where it might get tricky to some. Where does the code flow from here?!

You have to consider the fact that you are now inside of nested `if` statements and make note of where the beginning and end of all of those curly braces {} are. Allow me to re-write the code block with some comments that may help you visualize where the code will flow.

```
int age = 29;

if (age < 13)
{
    System.out.println("You are but a wee child!");
} // end if for age < 13
else if (age < 19)
{
    System.out.println("You are no longer a child, but a budding teenager.");
} // end else if for age < 19
else
{
    if (age < 65)
    {
        System.out.println("You are an adult!");
    } // end if for age < 65
    else
    {
        System.out.println("You are now a senior, enjoy the good life friends!");
    } // end if for nested else
    System.out.println("Also, since you are over the age of 19, you deserve a drink!");
} // end of final else
```

So, since the `age` is set to 29, the code will NOT execute the nested `else` condition. Because if it did, Java would be violating its most basic `if..else` rules for code flow right? So instead it will **skip** over the `else` block. And what do we see after the inner `else` block? Yet another console output line! So you will see this output on your console too: "Also, since you are over the age of 19, you deserve a drink!"

This gets outputted because we know we're inside that outer `else` block of code. Remember to follow through those curly braces {} to know exactly where you are.

The code then exits the outer `else` code block and the program will terminate.

Summary

Nested `if` statements are a fairly straight-forward concept that allows you to have good control over what code gets executed (and NOT executed) in certain scenarios. If you would like a good understanding of how Java reads through line by line, I'd recommend watching my video on Constructors. I will show you how the code flows in the debugger, and if you don't know how to debug yet, I'd also recommend watching this video!



Java Arrays

We've talked about the concept of Java Arrays back in Chapter 1, so if you like to you brush up on what we talked about, feel free.

What are Java Arrays?

As I've mentioned before, you can **think of an Array like a bunch of variables all smashed together under the same variable's name**. For example, you have a bunch of **contacts in an address book** and instead of creating variables to represent each contact like so:

```
String contact1, contact2, contact3, contact4 //etc...
```

You can combine them all into **one variable's name** like so:

```
List<String> contacts = new ArrayList<String>();
```

Now, this code is for an **ArrayList**, which for me, is used heavily in my day to day programming endeavors, but this is in fact just a flavor of Java Arrays as it's based on a Java **List**. Please allow me to introduce you to the original [Java Array](#).

```
String[] contacts = new String[10];
```

Okay, so, two things you'll notice:

- 1) The **square brackets []**
- 2) We've **put the number 10 in there**

The square brackets [] are Java's notation for an Array. They just signify that the variable type (in our example, String) will **not** be just a single variable but an **Array of those variables**. This means that you could throw those square brackets next to any variable type in Java to make them an Array.

Now let's talk about **why I put the number 10 in that code**. Well, one difference between an **ArrayList** (think no square brackets needed) and an Array (think square brackets needed) is that with an **Array**, you'll **need to define the size of the Array** when it's being initialized. What do I mean by that? Here's an example of code that would fail:

```
String[] contacts;
System.out.println(contacts[1]); // this will give you an error
```

The reason why this fails is because **you haven't initialized** the `Array`. So let's re-write this code with the initialization part included:

```
String[] contacts;  
contacts = new String[10];  
  
System.out.println(contacts[1]);
```

This will now work, but, you **won't see anything displayed in your console** because although you've initialized the `Array`, you haven't yet actually **populated it** with anything. How about we populate it with stuff? I want to keep this example simple, so I'm going to use a loop to populate the `Array`:

```
String[] contacts = new String[10];  
  
for (int i=0; i < contacts.length; i++)  
{  
    contacts[i] = "person" + i;  
    System.out.println(contacts[i]);  
}
```

What we've done here is **initialized** the `Array` with a size of **10** and we've created a "for loop" that will **iterate over each element** in the `Array` and populate it with something. In our case, we're iterating over the "for loop" 10 times, as the "for loop" is based on the size of the `Array` (which is 10 elements in length). And for each iteration, we are storing the String "person" followed by the **index** of the **current iteration** of the "for loop." It sounds complicated but here's the output:

```
person0  
person1  
person2  
person3  
person4  
person5  
person6  
person7  
person8  
person9
```

The word "person" is followed by the current index of the `Array` 10 times. Why did the loop start with `person0` and not `person1`? Weren't you expecting to see the numbers 1 through 10? Well, that happened for two reasons. The main reason it starts at zero is because my "for loop" was coded to start at 0 and run 10 times (i.e. 0 -> 9), but I did that for a reason. The reason I did it is because Java uses a **zero based numbering system** for its `Array`. When you want to refer to the **FIRST element in an array, you use the number 0**. The second element would be index 1. The third would be index 2 and so on. Had I programmed my "for loop" to start from index 1 and go until index 10, we **would have seen an error** when it tried to output the last element in the `Array` because element at index 10 doesn't exist! Only elements 0 -> 9 exist. One neat thing I'd like to point out about Arrays is that you can have something called a multi-dimensional Array. Now, having seen these in real world coding scenarios, they are a pain in the butt to understand and debug, so I wouldn't recommend using them unless you put

plenty of comments in your code describing its purpose. But for the sake of completion, I'll show you how to use them. Let's say you wanted to create a crossword puzzle program.

Example:

```
A D L E  
Q N N O  
W I N D  
V K N E
```

Here we have a 4x4 multi-dimensional Array. The code to create this crossword puzzle would look like:

```
String[][] crossword = new String[4][4];  
  
crossword[0][0] = "A"; crossword[0][1] = "D"; crossword[0][2] = "L"; crossword[0][3] =  
"E";  
crossword[1][0] = "Q"; crossword[1][1] = "N"; crossword[1][2] = "N"; crossword[1][3] =  
"O";  
crossword[2][0] = "W"; crossword[2][1] = "I"; crossword[2][2] = "N"; crossword[2][3] =  
"D";  
crossword[3][0] = "V"; crossword[3][1] = "K"; crossword[3][2] = "N"; crossword[3][3] =  
"E";
```

Now, having explained what an `Array` is in Java, I'd like to go back to what I mentioned before about the `ArrayList`. I find that the `ArrayList` is more useful in most coding situations, it has built in methods that allow you to do things like adding and removing elements **without having to worry about the size of the Array** (as it will grow and shrink as necessary). But, it is however, important to know what an `Array` is in Java and what the syntax looks like.



Primitive Data Types

What are primitive data types in Java? Well, you remember us talking about data types and Object Oriented programming right? Well, in Java, primitives are data types and have nothing to do with Object Oriented programming!

In Java, everything “extends” from Objects, except primitive data types. Think of primitives like the building blocks from which programming languages are built. If you were to read it in a catchy headline it would be “Programming 1.0 – Primitive Types” vs. “Programming 2.0 – Objects.”

What do primitive data types look like in Java?

```
int aPrimitiveInteger;
double aPrimitiveDouble;
char[] aPrimitiveString;
```

What you'll notice is that the primitive version of an `Integer` is `int`, the primitive version of a `Double` is `double` and the primitive version of a `String` is an array of `char`s (note: if you don't remember what an array is, [click here](#)).

Now that we know what these things look like, how do they work? Let's start by showing what the differences are... considering this code:

```
private static int anInt;
private static Integer anotherInt;

public static void main (String[] args)
{
    System.out.println(anInt);
    System.out.println(anotherInt);
}
```

What is the outcome of this code?

0
null

This is the first big difference between a primitive data type and an Object. Primitives have default values that are assigned to them as soon as they are declared (in all cases except when the primitives are [local variables](#)). In other words, as soon as I write the code: `private static int anInt`, the variable will have the value of 0. Now, I could have easily chosen to give it a value like this:

```
private static int aPrimitiveInteger = 32;
```

The point I'm trying to get at is that you don't have to give the primitive an initial value, for it'll have one as soon as it's created (again, in most cases). Primitive `int`'s are NEVER `null`. Whereas, their `Object` counterparts `Integer` always have a value of `null` if they're not assigned a value.

Why you should care about Primitive Data Types

Okay, so that's mildly interesting, but why do I really care about primitives? Well, the most compelling reason to use a primitive over its Object counterpart is performance. Primitives perform much better in Java than their Object counterparts. Let's say you have to write some code that will calculate the latitude and longitude of some given coordinates, but, those coordinates are in some other format (let's say UTM). There is a complicated formula for converting UTM coordinates into latitude/longitude coordinates. If you were to use nothing but Java Objects to convert these values, it would take MUCH longer to do so than if you were to use Java primitives. Now, this type of situation doesn't present itself often, but it's good to know this information anyway, because if you want to become a programmer in the real world, you'll need to be armed with this kind of knowledge. In the real world, whether you're writing code for your online blog, or for an international software behemoth, you want to create code that runs as fast as possible, because we all hate waiting on computers right?

Alright, so if primitives are faster than Objects, then why don't we always use primitives? Well, that's probably because their Object counterparts have lots of built in methods that help us do things. For example, you'll likely be in a situation when you may need to get the String equivalent of an Integer value, consider this method:

```
public void displayPhoneNumber(String areaCode, String number)
{
    System.out.println("(" + areaCode + ")" + number);
}
```

Now let's say all you have is Integers to pass into this method. You'll need to first convert your Integer into a String before Java will let you run your code, so you'll need to do this:

```
Integer myAreaCode = 416;
Integer myNumber = 5558262;

displayPhoneNumber(myAreaCode.toString(), myNumber.toString());
```

You see? We've converted our Integer to a String by using the `toString()` method of the Integer Object. If we had used a primitive `int`, we would not be able to do this.

So there's clearly a trade-off of performance vs. convenience. If you're dealing with straight up numbers and calculations, primitives are a solid choice; if you need to be manipulating your variables' data then

using the Object equivalent makes more sense. In any case, you should now have a better understanding of what primitive data types are in Java.



Strings

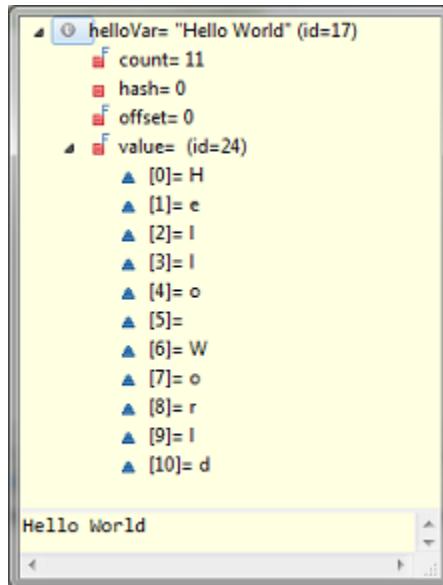
Strings in Java are a widely used data type (and in most programming languages), they allow us to store anything and everything that has to do with words, sentences, phone numbers, or even just regular numbers. But, as useful as they are, you'll need to know a few important things about Strings before you go wild using them in your code.

String - Object or Primitive?

Strings could be considered a primitive type in Java, but in fact they are not. A String is actually made up of an array of char primitives. In other words, if you create the String "Hello World" like so:

```
String helloVar = "Hello World";
```

then if you were to look at the helloVar variable in your debugger, you will see the following:



You can see how the String is made up of an array with 11 elements (0 -> 10) where [0] = H, [1] = e, [2] = l, [3] = l, [4] = o etc. Alright, so who really cares that it's made up of an array of chars then right!? Well, I just didn't want you to get confused when you saw your Strings in the debugger 😊

So let's move onto another important fact about Strings in Java, they are **immutable!** What does that mean? It means that once you create a String, it **cannot** be changed. So for the example above with my helloVar, if I wanted to change that variable's content, I would have to:

1. Re-assign a new value into the existing helloVar variable
2. create a new String object and assign the changed value to it

Okay, so you might be thinking “re-assign into the existing variable... wouldn’t that mean changing it? Which you just told me I couldn’t do!” Well, what happens behind the scenes is what’s interesting, but it will involve me talking about computer memory. Now this isn’t the most exciting of all topics, but I’ll have to explain it for completion sake, otherwise the professional programmers (also known as ‘geeks’) out there will start screaming for attention.

Memory

This topic is huge, and I could probably write a lot on this one topic, but to save you from that dull reading, I’ll give you a useful summary. You see, all variables in Java will be stored in memory on your computer. The variables will be assigned a location that is identified by an address. Think of the address in memory like your own home address. If you create a new `String` variable, it will be assigned an address that is currently empty. This is just like if you were to move into a new home, you’re obviously not going to move into a home where someone is already living right? Same thing happens with computer memory, Java will figure out where there’s some empty memory space and have the new variable “move in” or “occupy” that space. Now, since `Strings` are immutable, they cannot be changed, and thus, you cannot go to the address where that variable exists and starts messing around with it. The only way to change what currently exists at that address is to assign it a new address with its modified value. The comparison to the real world would be if you were living in your house at your address with your spouse and two children, but if you wanted to have a new child occupy space in your house, you’ll have to move into a new house first! It’s not the greatest example, but I feel like it works well enough. What if you want to just replace a `String` variable that already exists with a new value, what will happen then? Well, essentially Java will just demolish your house (clear your current variable’s memory) and move you into the house next door (assign the same variable to the next available memory location).

Phew, you see what I mean by a dull subject. All you really need to understand is that you can’t change a `String` in its memory address. Once it’s created, it’s there until it gets destroyed (also known as Garbage Collected... another huge topic that I won’t cover right now). So, let’s move onto something more interesting...

String Concatenation

One more fun thing you can do with Strings in Java is String concatenation. What does that mean? It just means that you can put a bunch of Strings together to make one bigger String. Here’s an example:

```
String str1 = "Hello ";
String str2 = "World";
```

```
String str3 = "!";
System.out.println(str1 + str2 + str3);
```

You see those plus + signs between the three variables in the `System.out.println`? That is string concatenation at work. What it's saying is to "append" the second string to the first, and then "append" the third string to the second. What's the result?

Hello World!

The usefulness of this may not be apparent at the moment, but one real world example I can think of is when you have your first and last names stored in two separate variables. You've probably seen this many times on the internet. You'll fill out a form of some sort to sign up for a mailing list perhaps... then you'll receive an email later that has your full name printed. This is an example of String concatenation. Example:

```
String firstName = "Trevor";
String lastName = "Page";

System.out.println("Hello there " + firstName + " " + lastName + ".");
```

This would output `Hello there Trevor Page`. You see how we not only put the plus + sign between the variables, but we also threw in some Strings that weren't assigned to variables? We can do this because whenever you put letters/numbers between those double quotes "", Java will interpret it as a `String` and will allow you to concatenate it to your other `String` variables! Neat 😊

The final thing I want to mention is that you can get yourself into some trouble when trying to compare two `Strings` to each other. For now I will just say that you need to use the `equals()` method whenever comparing two `Strings`. The reason for this is what I will be explaining in the next section, as it has to do with Objects in general.



Packages

What is a Java package? Why is it useful? What does it look like in code? These are the questions I'll answer!

What is a Java Package?

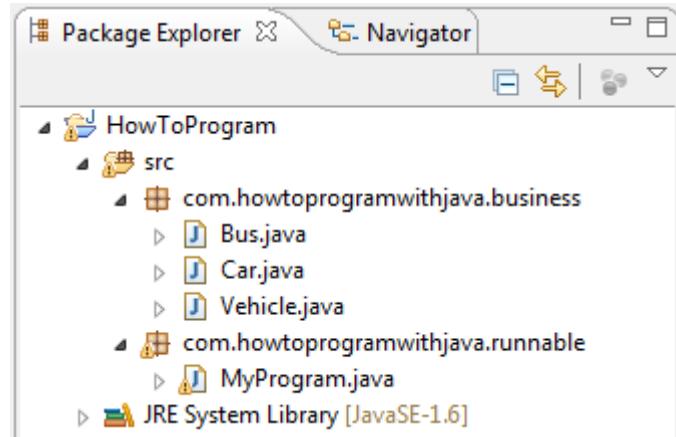
Well folks, Java packages are the means by which we **organize our code**. As a project grows and grows, we'll inevitably have more and more Class files (.java files). The way to think of a Java package is like a folder in your regular Windows/Mac file system. It is used to "package" together all of your **files into one logical place!** It's really just that simple boys and girls. 😊

Why are Java Packages Useful?

Don't get me wrong, it's entirely possible to not use any packages at all and just shove all of your project's files into one spot. But this is kind of like **storing all your music files and pictures right onto your desktop** without organizing them into folders... **it's just messy**. You can get away with it if you have a tiny program, such as a simple Java tutorial program. Anything bigger than that (let's say 5 Class files or more), then it might make sense to start creating packages.

What do Packages look like in Code?

Let's take a look at Spring STS and the way they display packages:



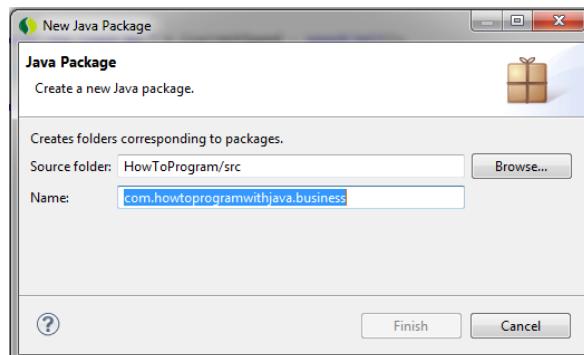
Alrighty, so here we see that Spring STS has a "Package Explorer" view. **If you don't see this in your Spring IDE, then try clicking on the following menus:**

- *Window -> Show View -> Package Explorer (or Other, then type in Package Explorer)*

So, what you see above is essentially the equivalent of having the following directory structure in your file system:

```
-> com
  -> howtoprogramwithjava
    -> business
      -- Bus.java
      -- Car.java
      -- Vehicle.java
    -> runnable
      -- MyProgram.java
```

If that makes sense to you, then your next question would probably be “Now that I understand **WHAT** a package is, **HOW do I create them in MY project?**” Piece of cake, all I did to create these packages was right click on my “src” folder (or the root How To Program folder) and choose *New -> Package*. You just need to type in your packages names separated by periods (.) like so:



By convention, package names should be **entirely lowercase**. They don't have to be, but it's just how everyone else does it, so you might as well be like the rest of the programming herd! Also, another convention is that if you're developing a Java project for a website, you generally **name your packages after the website address in reverse order**. My website is `howtoprogramwithjava.com`. In turn, the packages I would create are “`com.howtoprogramwithjava.someotherpackage`.” Again, just a convention, but it **makes it easier for any new programmer to jump into your code and understand everything with relative ease**.

What do Java packages look like in code?

Alright, so now that you have a good understanding of what these packages are all about, let's see how they translate in code. It's pretty simple really. Any Class file you've created that exists inside of a

package needs to have the **declaration of that package at the very top of the file**. So, in the case of my `Vehicle` Class file, the code will look like this:

```
package com.howtoprogramwithjava.business;

public class Vehicle
{
    // code removed for the sake of learning about packages only!
}
```

That's all there is to it, since the `Vehicle` Class file exists inside of the `"com.howtoprogramwithjava.business"` package, it needs to have that declared at the very top. And yes, it has to be at the top, otherwise your code won't compile properly. The first thing you'll see in any Java file is the package declaration, but what's tricky is that most Java tutorials online or in books leave this declaration out (simply because they assume that you know what a package is). So now I'll be the one to tell you that that's the case, I won't assume you know this 😊

Have I beat this concept into your heads now? Will you always remember what a package is, and that you should use them to keep things neat and organized? Do they seem straight-forward enough to you?

Imports

What are **Java imports**? How are imports used in Java code? Why do I need imports? Is there any easy way to manage imports? These are the questions I hope to answer in order to give you a full understanding of why these imports used.

What are Java Imports?

Well, since you are already familiar with Java packages, Java imports flow naturally from packages. In Java, there are TONS of useful **built in classes and methods** that allow us to do things like:

- Read the contents of a file / Create a file and populate it with contents
- Compare dates with each other (i.e. see if one date is before or after another)
- Send emails to anyone

Okay, that's great, so **what does that have to do with imports**? Well, all of these classes are nicely organized in packages, and if you want to USE these classes, you'll need to import them into your project. So, this means that before you can play around with Dates, you'll need to import the `Date` object first!

What do Java imports look like in Code?

Well, if we continue on our example of Dates, then it would look like this:

```
import java.util.Calendar;
import java.util.Date;

public class MyProgram
{
    public static void main(String[] args)
    {
        Calendar calendar = Calendar.getInstance();
        calendar.set(2012, 8, 19);
        Date firstDate = calendar.getTime();

        calendar.set(2012, 8, 1);
        Date secondDate = calendar.getTime();

        System.out.println("Is first Date before second Date? " +
firstDate.before(secondDate));
    }
}
```

```
        System.out.println("Is first Date after second Date? " +
firstDate.after(secondDate));
    }
}
```

If you have a keen eye, you'll spot in the **imports** at the very **beginning of this code segment**. This code means that in the package "java.util" there are Classes called "Date" and "Calendar," and we want to use them in our program! The reason we need to tell Java that we are going to use these Classes is for the sake of brevity and specificity. A great example of this is the `Date` class. You may or may not know this, but there are more than one `Date` classes in Java. There is a `java.util.Date` class and a `java.sql.Date` class.

You can imagine how it would be impossible for Java to decide which `Date` class you're actually referring to right, so you just need to **specify which one you'd like to use**.

Why do I need Java Imports?

Well, the truth is that you don't actually NEED to use imports. You could get away with referring to the **full package name** of the Class you wish to use. So the code above could be re-written like so:

```
public class MyProgram
{
    public static void main(String[] args)
    {
        java.util.Calendar calendar = java.util.Calendar.getInstance();
        calendar.set(2012, 8, 19);
        java.util.Date firstDate = calendar.getTime();

        calendar.set(2012, 8, 1);
        java.util.Date secondDate = calendar.getTime();

        System.out.println("Is first Date before second Date? " +
firstDate.before(secondDate));
        System.out.println("Is first Date after second Date? " +
firstDate.after(secondDate));
    }
}
```

Notice the **import statements at the top are gone** and we've now prefixed every `Date` and `Calendar` reference with `java.util`. This is the correct code and will compile just fine. But it's a bit **annoying** to have to write out the `java.util` every time you reference `Date` or `Calendar`. We have import statements in Java because it **makes life easier**!

Is there an easy way to use Java Imports?

There it is! In your Spring STS IDE, you can use a shortcut key that will automatically detect any Classes that haven't yet been imported and attempt to automatically determine what package that Class exists

in and put in the import statement in code. The shortcut keys are **Ctrl-Shift-O** (that's Control-Shift-“Oh,” not zero). Try it out yourself by copying/pasting the following code and you'll see that there are errors with all the `Date` and `Calendar` classes. Hitting **Ctrl-Shift-O**, Spring STS will ask you if you want `java.util.Date` or `java.sql.Date`. Choose `java.util.Date` and voila the import code is automatically added to the top of your Class file!

```
public class MyProgram
{
    public static void main(String[] args)
    {
        Calendar calendar = Calendar.getInstance();
        calendar.set(2012, 8, 19);
        Date firstDate = calendar.getTime();

        calendar.set(2012, 8, 1);
        Date secondDate = calendar.getTime();

        System.out.println("Is first Date before second Date? " + firstDate.
Before(secondDate));
        System.out.println("Is first Date after second Date? " +
firstDate.after(secondDate));
    }
}
```

Importing ALL Classes in a Package

There is one other “shortcut” method of importing classes in Java and that's by using a wildcard (*). Say for instance you just want to import ALL of the classes that belong in the `java.util` package, you could just use the code `import java.util.*`. This will import everything in `java.util`, but it's important to note that it won't import any sub-packages within `java.util`. However, this “importing method” is not really used much anymore due to the fact that IDEs (like Spring STS) are able to import classes automatically by organizing your import (i.e. via **Ctrl-Shift-O** shortcut key). But for the sake of completion, I figured I would talk about the wildcard imports. Use them if you like!

One final note

For those of you who may be asking the question, “How can I use things like `String` or `Integer` without having to do imports?” Excellent question and very astute observation! This is because the most commonly used Classes reside in the `java.lang` package and this package is essentially auto-imported for us for convenience. How nice of Java to do that for us! I hope that little Java tutorial helped clear the air on what Java imports are and how they are used.

Java Practice Assignment #1

[Click Here to Download Assignment Source Code](#)

Alright ladies and gentlemen, it's time to put your knowledge to the test!

Here's what we're going to do, I will outline the requirements for a practice assignment, and I will include an archive file with the contents of the assignment (at the top of this page). I will also include a video which will explain how to import the assignment into your SpringSource Tool Suite IDE and set it up so you'll be good to go.

Sound good?

The Requirements

The assignment is to simulate the lottery. You will need to implement code that will generate 6 lottery numbers between 1 and 49 (inclusive), you will then need to implement the code that will read in 6 numbers that you will type into the console yourself. Then the numbers you input will be compared against the randomly generated lottery numbers and it will output which numbers match (if any).

Here's the catch, you will need to make sure there are no duplicate numbers (either when being randomly generated or inputted in the console). It's just like a real lottery after-all!

What You'll Need to Know

Here's a list of concepts that you'll need to understand before you try to complete this assignment:

- Loops (for loop, while loop)
- Methods
- Variables
- Arrays (or Collections, preferably the `Set`)

Video Explanation on How to Import the Assignment

Here's a video that will explain how to import and setup the assignment into your IDE:

[Click Here for Video](#)



Chapter 4

Building Blocks



The Java Object

One of the most important and fundamental aspects to Java is the Java Object. Since this topic is central to the entire programming language, let's talk about the things you **SHOULD** know about Objects!

Okay, so what should you know? Well, first I want to talk about why the Java Object is so fundamental to the programming language.

Java Object

We've already talked about many examples of Objects in Java. These are what I would define as a "noun" in any sentence that would define a business problem. These "nouns" could include words like, **User**, **Library**, **Vehicle**, but what's one Object we haven't yet mentioned? The answer is easier than you may think. It's **Object**! Did you know that you could set a variable's type in Java to be `Object`? Well, you can... and it's quite handy!

Everything is an Object

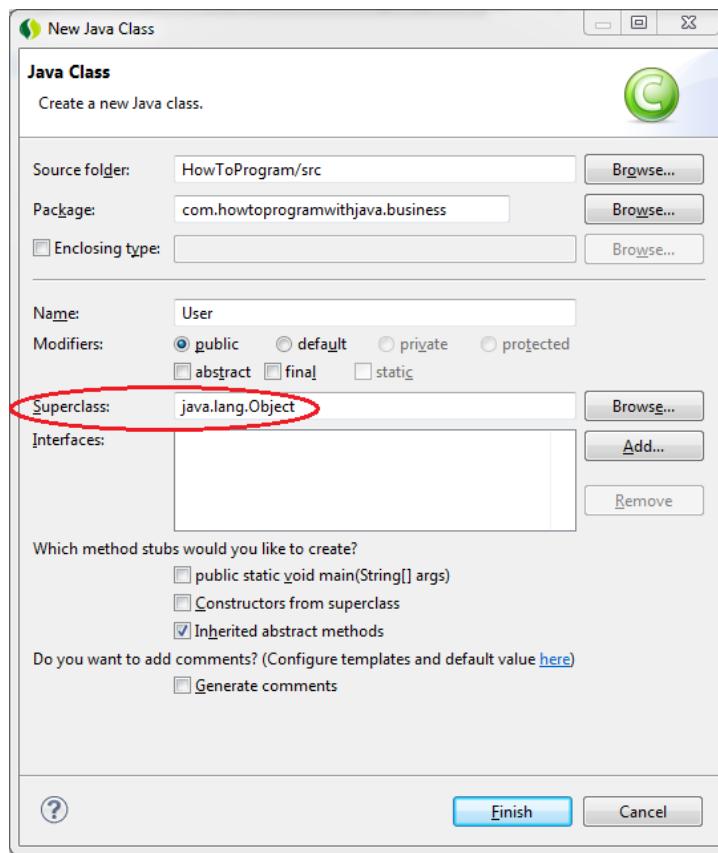
In Java, **any** object you create (`User`, `Library`, `Vehicle`) is actually of type `Object`. This concept is what is so fundamental to understand. So let's say you create the object `User`, it could look like this:

```
public class User
{
    private String username;
    private String password;

    public String getUsername()
    {
        return username;
    }
    public void setUsername(String username)
    {
        this.username = username;
    }
    public String getPassword()
    {
        return password;
    }
    public void setPassword(String password)
    {
        this.password = password;
    }
}
```

```
}
```

Now, there's nothing in this code that would indicate that this `User` object is actually of type `Object`. Usually, to determine if one particular object inherits properties of another object, you would look to see if your object **extends** another object. But in the case of our `User` there **is no extends** code. What the heck?! Well, since all Objects in Java **inherit from the Object type**, there's no need to explicitly show this in our code. If you don't believe me, try creating a new Object with your STS IDE. Here's a screenshot from when I made the `User` object above:



Notice that the super class of this `User` objects I'm creating automatically defaults to `Object`? Well, that's because all objects in Java inherit from `Object`!

Seriously, is everything an Object?

Okay so you believe me, everything inherits from `Object`, so you may ask "Are there any exceptions to this rule?" Well, obviously! Aren't there exceptions to every rule that exist in this crazy world?

The exceptions are primitive data types. Primitives do not inherit from the `Object` type in Java, but that's the only exception to the rule. Everything else in Java either directly (or indirectly) inherits from the `Object` type.

What are the implications to inheriting from Object?

What's interesting to note is that since every object in Java inherits from the `Object` type, that means that whatever methods are defined (as `public` or `protected`) on the `Object` type should be available to any object that WE create, right? Right!

So what are these methods?

Excellent question, let's take a look at the `User` object we created above. If we were to instantiate this `User` and take a look at which methods could be invoked on the `User`, what would we see? Well, we would expect to see the `public` methods that we defined right? These would be:

```
setPassword()  
getPassword()  
setUsername()  
getUsername()
```

But here's what we **actually** see:

- `equals(Object obj) : boolean - Object`
- `getClass() : Class<?> - Object`
- `getPassword() : String - aUser`
- `getUsername() : String - aUser`
- `hashCode() : int - Object`
- `notify() : void - Object`
- `notifyAll() : void - Object`
- `setPassword(String password) : void - aUser`
- `setUsername(String username) : void - aUser`
- `toString() : String - Object`
- `wait() : void - Object`
- `wait(long timeout) : void - Object`
- `wait(long timeout, int nanos) : void - Object`

There are a few methods there that we didn't create, these include:

```
equals()  
getClass()  
hashCode()  
notify()  
notifyAll()  
toString()  
wait()
```

```
wait(long timeout)
wait(long timeout, int nanos)
```

And if you notice in the screenshot above, on the far right of all these methods, is the word `Object`. This is because all of these methods belong to the `Object` type. Make sense? It's nice to know what's actually going on here. The only other topic to cover is what all of these methods are used for, but I don't want to dive into all of them in detail, so I'll get the ones I don't want to talk about out of the way right now.

Threading Methods

- `notify()`
- `notifyAll()`
- `wait()`
- `wait(long timeout)`
- `wait(long timeout, int nanos)`

If you've ever heard of the term "Multi-threading," it has to do with being able to simultaneously run multiple "tasks" all at once. The above methods help facilitate multi-threading. Although they are very interesting in their own rights, I don't think it would be useful to talk about them at this point. Let's look at the methods that remain:

equals()

This is a method you've probably seen before. We use it when comparing two `String`s to each other. It's the method that we use when we compare **any** two objects with each other. You're probably saying hang on a second, I thought we used the "`==`" operator to compare objects! You've seen it before:

```
int someNumber = 478;
int someOtherNumber = 983;

if (someNumber == someOtherNumber)
{
    System.out.println("These numbers are equal");
}
else
{
    System.out.println("These numbers are NOT equal");
}
```

Why are we talking about a `equals()` method then? Well, remember when I said that primitive types are the exception to the rule? That means that primitive types don't inherit from the `Object` type right? If that's the case, then that means that primitive types don't have access to the methods that are defined within `Object`, which includes `equals()`!

Very interesting, so then if we want to compare two objects, then we just use the `equals` method then right? This is true but there's a bit of work that we'll need to do to ensure that when we **do** compare two objects, we get the expected result. Consider the following:

```
User user1 = new User();
User user2 = new User();

user1.setUsername("Trevor Page");
user1.setPassword("aPassword");

user2.setUsername("Trevor Page");
user2.setPassword("aPassword");

System.out.println("Are users equal? " + user1.equals(user2));
```

What would you expect to see as the output here? We are instantiating two `User` objects, and we are assigning the exact same username and password to both `Users`. Then we are just invoking the `equals()` method to check if both `Users` are equal. We would assume that since both the username and password are equal and that the result would be “true.” Here’s the actual result:

```
Are users equal? false
```

Well what the heck! Why aren’t they equal? Well, this comes down to inheritance and the default implementation of the `equals()` method.

Now pay attention because this is important!

If you do not override the default implementation of the `equals()` method, then Java will default to the strictest implementation of an `equals` comparison. And that is the “`==`” operator!

What does the “`==`” operator do?

The “`==`” operator (spoken as “equals operator”) compares two objects by their physical memory address.

So how does this apply to our example above with the two `Users`? Since we haven’t overridden the `equals()` method, it will compare the two objects using the “`==`” operator, which will compare their addresses in memory. Since I created two objects (via the `new` keyword), this means we have two separate objects in two separate memory locations. Since they have two unique locations in memory, the `equals()` comparison will return `false`.

How do we override the `equals()` method?

In our `Users` example, we would like to compare two users to each other, but we don’t want to compare the physical memory addresses. We want to define our own meaning of equals, so what in your mind would constitute equality among `Users`? I think it would just be if two `Users` had the same username!

Hopefully in any web application no two users would be allowed to have the same usernames. If we have two `User` objects and they both have the same username, then we could consider them equal.

How do we accomplish this? We override the `equals()` method on the `User` object! So our new `User` object would look like this:

```
public class User
{
    private String username;
    private String password;

    public String getUsername()
    {
        return username;
    }
    public void setUsername(String username)
    {
        this.username = username;
    }
    public String getPassword()
    {
        return password;
    }
    public void setPassword(String password)
    {
        this.password = password;
    }

    @Override
    public boolean equals(Object obj)
    {
        return this.username.equals(((User) obj).getUsername());
    }
}
```

Notice the last section of our `User` object has an `@Override` over the `equals()` method. Well, this is how we override the `equals()` method of the super class (the `Object` class). This is the inheritance I was talking about. In Java we are allowed to “change” the behavior of a method that we inherit from a parent Class. In this case, the parent Class of our `User` object is `Object`. Because the `Object` Class has an `equals()` method, then that means we can change its behavior in our child class (the `User` Class). Since we didn’t like how the `Object`’s `equals()` method worked, we override its functionality in our `User` Class! And this is the code you see above.

So! This is fairly complex stuff if you’re new to programming, so allow me to keep explaining. Let’s take a look at the code where we actually compare the two `Users` together, as this may shed some light on this subject...

```
User user1 = new User();
```

```
User user2 = new User();

user1.setUsername("Trevor Page");
user1.setPassword("a Password");

user2.setUsername("Trevor Page");
user2.setPassword("a Password");

System.out.println("Are users equal? " + user1.equals(user2));
```

This code will now output what we would expect:

Are users equal? true

Okay great, so maybe you're thinking: "I kind of understand the concepts, but I really don't understand how that's reflected in the code!" Let me try to explain what's happening.

I want you to concentrate on the most important part of the code above:

```
user1.equals(user2)
```

Remember that the `User` definition (with the `@Override equals` code) is a Class that represents the blueprint for any `User` object. So when we **instantiate** a `User` object, each one will essentially have its own version of the `@Override equals` method defined in the `User` Class.

This means that when we run that important code, we are saying:

Hey! `user1`! Run your `equals()` method, and pass in `user2` as a parameter. So what we may actually see if we were to debug this is the following (I've commented out the actual code, and instead, replaced the code with what would essentially be replaced at runtime when Java runs the code):

```
@Override
//public boolean equals(Object obj)
public boolean equals(User user2)
{
    // return this.username.equals(((User)obj).getUsername());
    return user1.username.equals(user2.getUsername());
}
```

So, what really ends up happening is that we pass in the `user2` object as a parameter. Now, what normally happens is the `equals` method takes an `Object` as a parameter, but since a `User` is an `Object`, Java is okay with you doing this (inheritance). Then we say, `this.username`. Well `this` just refers to whatever Object the Class represents. So in our case, since we invoked the `equals()` method of `user1`, that means we're inside of the `user1` `equals` method, which means that when we say `this` we really mean `user1`! But then we say `this.username.equals()`, doesn't that mean we just invoke the `equals` method on `user1` again, resulting in an endless loop? No! We're invoking `equals` on `user1.username`...

and `username` is just a `String`. So that's perfectly legitimate, as `String` defines its own `equals()` method. Then we just pass in `user2.username` into the `equals()` method for the `String` comparison. Which we know that, if both `Strings` are the same, then it will return true!

Phew, that's some hardcore coding there! If you understand it, then that's amazing, and I've done a great job of explaining the concepts. If you don't understand it, then you're probably part of the large portion of the new programmers on this planet. The topics that stem from Java's `Object` class are quite complex as they require a solid understanding of Object Oriented Programming (more specifically inheritance and polymorphism). So, if you don't get it, take some time to re-read this tutorial and maybe try messing around with the coding examples I've given.

Loose ends

I haven't talked about the other methods that come from the `Object` class, such as `hashCode()` and `toString()`. So let me touch on these quickly.

`toString()`

This method is used to return a `String` representation of our objects. The default implementation of this (from the `Object` class) will just return a readable version of the physical memory address. So if I were to do a `System.out.println()` on the `user1` object, I would get the following:

```
com.howtoprogramwithjava.business.User@7d8a992f
```

This means nothing to us, and it's not helpful, other than if you would like to check to see if two objects reference the same memory location. So that's why we can override the `toString()` method to give something more meaningful. Let's override it! Please add the following method to our `User` class

```
@Override  
public String toString()  
{  
    return "Username: " + this.username + ", password: " + this.password;  
}
```

Now, if we were to invoke `user1.toString()`, we wouldn't get a meaningless representation of a memory location, we would get:

Username: Trevor Page, password: a Password

Hey! That actually makes some sense and it's useful! So, that's a quick rundown on the `toString()` method.

`hashCode()`

This is a more advanced topic that revolves around the use of `HashMaps`. So for the sake of completion (and for the sake of not overloading your brain), I'll just say that generally when you override

the `equals` method, it's good practice to also override the `hashCode()` method. This is so that if your objects are put into a `Map`, Java will be able to "store" them nicely inside of your `HashMap`. How do you override `hashCode()`? Also slightly advanced, but for now try this... In your STS, try right-clicking in your User code then:

Source -> Override/Implement Methods -> Choose Hash Code -> OK

Summary

I encourage you to take a break, try to digest this information, and perhaps come BACK and re-read everything so you have a good understanding of these concepts. As I've said, these are more advanced topics, but they are **critical** to understand if you want to get the hang of programming in Java.

In Object Oriented programming (i.e. the Java programming language), Inheritance is one of the key principles that is beneficial to use in the design of any software application. Java inheritance allows for a neat way to define relationships between your Objects (an in turn re-use your code so you don't have to type the same stuff over and over again).



What is Java Inheritance?

What do I mean when I say that Inheritance allows you to **define relationships between Objects**? Well, let's think of some examples of Objects that DO have one or more relationships. Think the object `Vehicle`, for this is a fairly generic term for:

1. Car
2. Bus
3. Motorcycle



Do you see how a Car *is a* Vehicle, how a Bus *is a* Vehicle, how a Motorcycle *is a* Vehicle etc.? This *is a* relationship and is what Java Inheritance is all about. When you can verbally say that *something is a something else*, then you have a relationship between those two Objects, and therefore you have Inheritance.

How does Inheritance help us?

Well, with the examples given above, this means that a Car inherits behaviors and/or attributes from a Vehicle. So let's think about this for a second, what is a Car? Well, it's a Vehicle with four wheels, doors and around 5 seats. Okay, what's a Bus? It's also a Vehicle likely with more than 4 wheels, doors and probably somewhere around 30 seats. What's a Motorcycle? It's a Vehicle with two wheels, no doors and one or two seats. Once you start to "map" out all of the characteristics of your Objects, you'll begin to see what similarities they have (i.e. what they have in relation), and also what they **don't** have in relation to each other. This is very important with Java inheritance. If all your Objects share something in common, then this can be considered an **attribute of the super class**. Whatever they do not have in common will be **attributes of the child classes**.

What is a super Class and what is a child Class?

With our example, the super class is the Vehicle object and the child classes are the Car, Bus and

Motorcycle. The super Class is essentially the Object that will hold all the attributes that are common, so our Vehicle super Class would have the following attributes:

- Wheels
- Seats

From our inspection of all the types of Vehicles above, we identify that all types of Vehicles have wheels and seats. But notice that I didn't put doors as part of the Vehicle object. This is because Motorcycles don't have doors! Doors would only be attributes of Cars and Busses, so we will have a door attribute on the Car and Bus Objects. Make sense?

Coding Inheritance in Java

When we're coding this thing called Inheritance in Java, what does it look like? Well it can take the form of either **or an Abstract Class**.

- **Interface** = An outline (or skeleton) of an Object with **no** implementation
- **Abstract Class** = An outline of an Object that **can** contain an implementation

Without further delay, let's look at some examples of an **Interface** and an **abstract class**.

Interface

```
public interface Vehicle
{
    public Integer getNumberOfSeats();
    public Integer getNumberOfWheels();
    public String getVehicleType();
}
```

Here we've declared an Interface for our `Vehicle` and it has three methods, `getNumberOfSeats()`, `getNumberOfWheels()` and `getVehicleType()`. As you can see, there is no **implementation** of the code, we've just outlined some methods. So, now to make this interface useful, we need to implement it somewhere! So let's see what that would look like:

```
public class Car implements Vehicle
{
    @Override
    public Integer getNumberOfSeats()
    {
        return 5;
    }

    @Override
    public Integer getNumberOfWheels()
    {
        return 4;
    }

    @Override
    public String getVehicleType()
    {
        return "Car";
    }
}
```

```

    }

    public Integer getNumberOfDoors()
    {
        return 2;
    }
}

```

This is what the `Car`'s implementation of the `Vehicle` interface would look like! For this example, I've stated (in code) that a `Car` has 5 seats, 4 wheels and 2 doors. Let's see what a `Bus` would look like:

```

public class Bus implements Vehicle
{

    @Override
    public Integer getNumberOfSeats()
    {
        return 35;
    }

    @Override
    public Integer getNumberOfWheels()
    {
        return 6;
    }

    @Override
    public String getVehicleType()
    {
        return "Bus";
    }

    public Integer getNumberOfDoors()
    {
        return 4;
    }
}

```

This is self-explanatory right? Well, except for those `@Override` lines. What do those mean? These are called **annotations** and these were introduced in Java version 5 (we are currently on Java version 7). An annotation is anything that you see with an @ (at) symbol before some text above a method declaration or a class declaration. These particular `@Override` annotations are just saying that the method below it is from a parent (or super) class, and we are implementing the desired behavior in this particular class. In the case of an Interface, we **have to override the methods**, as it's a requirement with Interfaces.

Take special note that we don't have an `@Override` annotation on our `getNumberOfDoors()` method. This is because it wasn't declared in our Interface. Remember why? It is because a `Motorcycle` doesn't have doors, so it wouldn't make sense to put it in the Interface! Now, don't get me wrong, in the world of programming there are always several ways to solve the same problem, so you could have put something like `hasDoors()` in the `Vehicle` Interface and had it return a `Boolean` value of true or false (true in the case of `Car` and `Bus`, false in the case of `Motorcycle`). But, for the purpose of

illustrating that you can have your own non-overridden methods in your child classes, I chose to do it the way I did it.

Abstract Class

Let's look at abstract classes now. Remember that abstract classes don't necessarily need their methods overridden, and the methods can contain implementation if you want. If we were to create an abstract class for the `Vehicle` object, it could look like this:

```
public abstract class Vehicle
{
    public String vehicleType;

    public Integer getNumberOfSeats()
    {
        if (this.vehicleType.equals("Car"))
        {
            return 5;
        }
        else if (this.vehicleType.equals("Bus"))
        {
            return 20;
        }
        else if (this.vehicleType.equals("Motorcycle"))
        {
            return 1;
        }

        // the vehicleType variable has not yet been set to anything,
        // so we cannot say what number of seats this vehicle has, so
        // we will return null.
        return null;
    }

    public String getVehicleType()
    {
        return this.vehicleType;
    }

    public abstract Integer getNumberOfWheels();
}
```

As you can see here, we have some real code implemented in our `getNumberOfSeats()` method. The code relies on the `vehicleType` attribute. Let's take a look at how a child class would use this `Vehicle` abstract class:

```
public class Car extends Vehicle
{
    public Car ()
    {
        this.vehicleType = "Car";
    }
}
```

```
public Integer getNumberOfWheels ()  
{  
    return 4;  
}
```

The first noticeable difference between an interface and an abstract class is that you need to use the keyword `implements` when you want a child class to use an Interface, and you need to use the keyword `extends` when you want a child class to use an abstract class. We've also done something interesting with this code:

```
public Car ()  
{  
    this.vehicleType = "Car";  
}
```

This is called a **constructor**. The purpose of a constructor in Java is to outline a section of code that will be **executed when an Object is first instantiated**. So, this just means that when someone creates an instance of our `Car` Object, Java will automatically set the `vehicleType` to be "Car."

So now, if we were to write some code get the number of seats that our `Car` has, we would see that it has 5, because Java will see that the super class (`Vehicle`) has a method that defines the number of seats (`getNumberOfSeats()`).

Okay, so now for those who want to go the extra mile, I challenge you to put together a Java program that will allow you to use an abstract `Vehicle` class and properly output the following console lines:

```
My Car has 2 seats.  
My Car has 4 wheels.  
My Bus has 20 seats.  
My Bus has 6 wheels.
```

Using this Java main class:

```
public static void main(String[] args)  
{  
    Vehicle myCar = new Car();  
    System.out.println("My " + myCar.getVehicleType() + " has " +  
myCar.getNumberOfSeats() + " seats.");  
    System.out.println("My " + myCar.getVehicleType() + " has " +  
myCar.getNumberOfWheels() + " wheels.");  
  
    Vehicle myBus = new Bus();  
    System.out.println("My " + myBus.getVehicleType() + " has " +  
myBus.getNumberOfSeats() + " seats.");  
    System.out.println("My " + myBus.getVehicleType() + " has " +  
myBus.getNumberOfWheels() + " wheels.");  
}
```


Java Practice Assignment #2 – Employees

Before I get into the details of your next assignment, I'll show you my solution to the first assignment for you to look over and understand.

[Click Here to Download Practice Assignment 1 Solution](#)

Here's a video with a full walk-through of my solution:

[Click Here for Video](#)



Assignment 2 – People, Employees and Organizations

[Click Here to Download Practice Assignment 2](#)

My goal for this assignment is to get you familiar with inheritance and the importance of dealing with the `public` methods available in the Java Object class. In practice assignment 2, you'll learn how to use both an `interface` and an `abstract class`.

Important Notes:

1. I've included two library files (JAR files) in the source code of this assignment. You still need to add them to the classpath when you extract this assignment and begin working on it. To do this, just right click on the Project and select Properties, then Java Build Path, then "Add JARs", and navigate to the "src/lib" directory to add both JAR files.
2. As with the first assignment there are Tests available that must pass. Currently they should all fail AND they will have compilation errors. The compilation errors are expected, as you'll need to implement the appropriate methods from the interface and abstract classes (and then some). Once you've successfully coded the assignment, you won't have any compilation errors and all the tests will pass. To run the tests, just right click on the "Tests" class name and select "Run As->JUnit test".

Assignment Requirements

Okay so here's the breakdown of the requirements for this assignment. You will need to develop a system that can track employee information for two Organizations (Google and Microsoft). The Employee information you must track is as follows:

- Name
- Sex
- Job Title
- Organization they work for
- Birthday

As for the Organization that the Employee works for, you must also track this information:

- Organization Name
- Number of Employees

The system must be able to properly compare any two employees against each other to determine if they are the same Person. This means that if you compared two People with the same Name, Sex, Job Title and Organization, the system should think that they are equals to one another. If any of these properties are different, then the two People are not the same Person.

The same rules apply to comparing Organizations to one another. Organizations with the same Organization name are to be thought of as equal, different names means different organizations.

The Static Keyword

You've seen me use this `static` keyword every now and then in my coding examples, but I've never really explained what it meant. Perhaps you've been wondering what it meant, and perhaps you scoured the web for an explanation only to come up with definitions like this one:

In computer programming, a static variable is a variable that has been allocated statically — whose lifetime extends across the entire run of the program. This is in contrast to the more ephemeral automatic variables (local variables), whose storage is allocated and reallocated on the call stack; and in contrast to objects whose storage is dynamically allocated.

What!? First of all, I hate it when a definition of a term USES that same term IN the definition. So, if you don't understand what that stuff means, no worries, I'm here to help you!

What does Static mean in Java?

The static keyword is related to a Class. Do you remember what a Class is? It's the blueprint for an Object. When you write the code for a Class, that code is the "be all and end all" guide for how an Object lives its life in your program. In order to understand the static keyword, it's very important that you understand the difference between a Class and an Object. Let me ramble about that difference a little bit.

An Object can be instantiated. This means that it is essentially "brought to life" in your program. An example of the instantiation of an Object could be:

```
Vehicle car = new Car();
```

We have just instantiated a `Car` Object based on the `Vehicle` super class. So what does this imply? Well, we had to create two classes for this code, a `Vehicle` Class and a `Car` Class (a `Car is a Vehicle`). Somewhere in the computer's memory, a `Car` Object will exist. What's critical to note here is that BEFORE this `Car` was *instantiated*, there was nothing in the computer's memory, there was just two Class files (or blueprints) for a `Car` and a `Vehicle`. This is the critical difference between a Class and an Object. One exists in memory where you can access it and change it (the Object), and the other exists in part of the memory that you can't access as a code blueprint FOR an Object (a Class).

Okay, so now that I've drilled that into your brains, I can talk about what the `static` keyword in Java means. In Java, you can make variables and methods `static`. Static methods and variables can also be called *static methods* and *static variables*. Why they are also called this? It is because a static method (or variable) exists as part of the Class and **not** the instance of that Class (the Object). The implication of this

is that you don't need to instantiate an Object of that Class in order to access the static method/variable that belongs to that Class. Does that make sense to you? Perhaps not entirely, so let's use some real examples! Static methods are often used as "helper" methods. These helper methods are nice, because you don't have to instantiate the Object in order to use them. Let's say there's a global speed limit of 80, and you want to be able to check to see if you're over the speed limit at any given moment in time, your code could look like this:

```
// this would be inside a Vehicle.java file
public abstract class Vehicle
{
    public static Integer speedLimit = 80;

    public static void checkIfOverSpeedLimit(Integer currentSpeed)
    {
        if (currentSpeed > speedLimit)
        {
            System.out.println("Slow down! You're over the limit by " + (currentSpeed - speedLimit));
        }
        else
        {
            System.out.println("Go faster, you're only going " + currentSpeed);
        }
    }
}

// this would be inside a MyProgram.java file
public class MyProgram
{
    public static void main(String[] args)
    {
        Vehicle.checkIfOverSpeedLimit(70);
        Vehicle.checkIfOverSpeedLimit(155);
    }
}
```

So when we run this program (from the `MyProgram` -> `main` method) we'll see the following output:

```
Go faster, you're only going 70
Slow down! You're over the limit by 75
```

So, this may not seem like we've done something different from any other examples I've given you, but there's one main difference. We've called a static method on the `Vehicle` Class:

```
Vehicle.checkIfOverSpeedLimit(70);
Vehicle.checkIfOverSpeedLimit(155);
```

Notice how this is a little different from how we usually call methods. There's no instantiation going on here! Normally we would have to do this:

```
Vehicle car = new Car();  
car.checkIfOverSpeedLimit(70);
```

If you were to type this code into your `MyProgram -> main` method, you'll see that it gets highlighted with a yellow underline and you get the following warning:

The static method `checkIfOverSpeedLimit(Integer)` from the type `Vehicle` should be accessed in a static way.

Your IDE is yelling at you because you're trying to invoke a static method (Class method) on the actual instance of the Class (the Object). Now, mind you, this will compile and it will run properly, it's just not a recommended approach, as its static and should be used in the static way, which is to use the Classes name followed by the static method (or variable). For example,

`Vehicle.CheckIfOverSpeedLimit(Integer)`. Now, there's one little thing you'll need to remember, and that's the fact that you cannot use an instance variable (non-static variable) inside of a static method. This is because you're trying to ask the blueprint (the Class) to perform some sort of operation with a variable that may not even exist yet! Remember that instance variables (non-static variables) are tied to the actual instance (or instantiated version) of their Class. So let's say you had 6 Car objects, 3 Bus Objects and a Motorcycle Object. If you use an instance variable in the static method, how the heck is the program supposed to know what value to put into the variable? It can't know! A good way to think of this is that when you declare a static variable on a Class, ALL the Objects that get instantiated from that Class will *share* that variable. If you're familiar with programming already, this is like a global variable, as all instances of that Class will be able to access the same variable 😊



Constructors

So, what are Constructors? Why are they important? What do they look like in Code? Let's find out!

So, what are they? A constructor is really the building blocks of an `Object`. It's the first thing that Java will execute when a new `Object` is instantiated. If you were to instantiate a new `Object`, Java will go to that `Object`'s constructor code and execute whatever you had placed in that code. Make sense? It is pretty straightforward.

Why are Constructors important?

Well, it allows us to really initialize our `Objects`. It puts them into a default/known state, and this is really important when you have an `Object` that should be in a default state. You don't want to have variables that are left as `NULL`. You want to immediately put those variables in a known state in that `Object`, so you won't have any errors or strange behavior down the line.

What do Constructors look like in Code?

Let's find out. I've put together a little example that revolves around `Animals` and `HumanBeings`. It uses a little bit of Inheritance – a `HumanBeing` is an `Animal`.

HumanBeingProgram.java

```
package com.howtoprogramwithjava.constructors;

public class HumanBeingProgram extends Animal
{
    public static void main (String[] args)
    {
        HumanBeing me = new HumanBeing ();
        output(me);

        HumanBeing you = new HumanBeing ("blue", "female", "Jane Doe");
        output(you);
    }

    private static void output(HumanBeing human)
    {
        System.out.println(human.getName() + "'s eyes are: " + human.getEyeColor());
        System.out.println(human.getName() + " is " + human.getSex());
        System.out.println("-----");
    }
}
```

```
    }
```

HumanBeing.java

```
package com.howtoprogramwithjava.constructors;

public class HumanBeing extends Animal
{
    String name;

    public HumanBeing ()
    {
        super ();
        this.name = "John Doe";
    }

    public HumanBeing (String eyeColor, String sex)
    {
        super (eyeColor, sex);
    }

    public HumanBeing (String eyeColor, String sex, String name)
    {
        super (eyeColor, sex);
        this.name = name;
    }

    public String getName ()
    {
        return name;
    }

    public void setName (String name)
    {
        this.name = name;
    }
}
```

Animal.java

```
package com.howtoprogramwithjava.constructors;

public abstract class Animal
{
    String eyeColor;
    String sex;

    public Animal ()
    {
        this.eyeColor = "brown";
        this.sex = "male";
    }
}
```

```

public Animal (String eyeColor, String sex)
{
    this.eyeColor = eyeColor;
    this.sex = sex;
}

public String getEyeColor()
{
    return eyeColor;
}

public void setEyeColor(String eyeColor)
{
    this.eyeColor = eyeColor;
}

public String getSex()
{
    return sex;
}

public void setSex(String sex)
{
    this.sex = sex;
}

```

As you can see here, a constructor looks like a method, but it's missing one key part that defines a method. The return type! A constructor looks just like a method with no return type, but the constructor is named after the Class it belongs to. If we even change the "case" of a constructor's name, we'll get an error. For example, if we change the `Animal` Class' constructor to say:

```

public Animal ()
{
    this.eyeColor = "brown";
    this.sex = "male";
}

```

Then Java will complain and say "the return type is missing" because it thinks you're trying to create a method. Since we're not creating a method, we're creating a constructor, so it has to be exactly the same name as the Class that defines it.

You can have multiple constructors in a Java class and this is accomplished by using different parameters in your constructors.

Now, when you instantiate a Class, Java will execute the constructor that you specify – which is determined by the parameters that you pass in, so if you were to execute this code:

```
HumanBeing me = new HumanBeing();
```

Java will execute the constructor for Human Being that does not have any parameters/arguments. So this would execute the following code:

```
public HumanBeing ()  
{  
    super();  
    this.name = "John Doe";  
}
```

Now, there's some new code there. That `super` keyword.

What the heck does `super` mean? This just means that we want to call the constructor that is one step up in our inheritance chain. This means that Java will call the `Animal` constructor. Now, which one will it call? It will call the constructor that takes NO parameters/arguments because we have not specified any parameters in our `super()` code! Here's the code it'll end up executing:

```
public Animal ()  
{  
    this.eyeColor = "brown";  
    this.sex = "male";  
}
```

So, when we run this Java application, what is the output that we should expect to see? Well, here's the output that this program creates:

```
John Doe's eyes are brown  
John Doe is male
```

```
_____  
Jane Doe's eyes are blue  
Jane Doe is female
```

Does that make sense to you guys? Constructors are fairly straight-forward and this is pretty much as complicated as I can think to throw at you with the `super()` keyword and having multiple constructors with different parameters. Normally you just see constructors with no parameters that just initialize some variables, but you'll see a bunch of different examples of constructors as you go through your Java training or career as a Java programmer!

This Keyword

You may have noticed a special `this` keyword being used, and I'm sure you were wondering what it meant. Well, to understand the `this` keyword, you'll need to make sure you understand the difference between a Class and an Object. We talked about the difference [here](#).

I'll assume that you **do** understand the difference between a Class and an Object, so all I need to tell you is that the `this` keyword is what an Object uses to refer back to itself. Some other programming languages use the keyword `self`, when an Object is referring to itself. Java simply uses `this`.

Why would an Object refer to itself?

Well, I'd say it's both out of **necessity** and for the sake of **keeping things neat**. Here's what I mean:

Using `this` out of Necessity

Let's say we're working in a scenario where an Object has two constructors, one has two input parameters, and the other constructor has three input parameters. Now, what would happen if one constructor had to call the other constructor? How would you do it?

You may be able to make a lucky guess and say that you would use the `this` keyword. Lucky guess! Well how about I **show you an example** based on one that you have already seen, the `HumanBeing`:

```
package com.howtoprogramwithjava.constructors;

public class HumanBeing extends Animal
{
    String name;

    public HumanBeing (String eyeColor, String sex)
    {
        this(eyeColor, sex, "John Doe");
    }

    public HumanBeing (String eyeColor, String sex, String name)
    {
        super(eyeColor, sex);
        this.name = name;
    }

    public String getName()
    {
        return name;
    }

    public void setName(String name)
    {
        this.name = name;
    }
}
```

```
}
```

As you can see, in the first constructor of the `HumanBeing` Class, we are calling the second constructor with a default value for the `name`. We are able to do this by using the `this` keyword. You see, constructors are only used when an Object is instantiated/brought to life. The `this` keyword is only valid when you're dealing with an Object that has been instantiated, as it doesn't make sense to have a non-instantiated Class be able to refer to itself right? If the Class hasn't been brought to life, then how could it be pointing to itself... technically it doesn't even exist!

So, since we do have an instance of a `HumanBeing` object, then it makes sense that it could refer to itself. So let's say we instantiate a `HumanBeing` without a name. What would happen?

```
package com.howtoprogramwithjava.runnable;

import com.howtoprogramwithjava.constructors.HumanBeing;

public class MyProgram
{
    public static void main(String[] args)
    {
        HumanBeing me = new HumanBeing("Brown", "Male");

        System.out.println("My name is: " + me.getName());
    }
}
```

Here you see that we just created a `HumanBeing` object with the variable name `me`. We instantiated it using just two input parameters (`"Brown"`, `"Male"`), this means that the code will flow into the first constructor, and then invoke its own (second) constructor by passing in a third input parameter of `"John Doe"`. Neat right? Here's the output from the code:

```
My name is: John Doe
```

Using `this` to keep things neat

The other case when the use of `this` is handy, is when we are setting the values of some instance variables (those are variables that belong to each individual instance of an Object, which means the values can change from Object to Object). When you are inside of a setter method, the variable names of the **input parameters will take precedence over the instance variable names**. This may sound confusing, but the concept is pretty simple. Let's use the setter method for the `HumanBeing`'s name:

```
public void setName(String name)
{
    this.name = name;
}
```

There's a reason why we need to say `this.name = name` and that's because of scope of variables. You see, when the code is flowing **inside** of this **setter method**, it has a reference to the variable `name`, because it's being passed in as an **input parameter**. So now, while the code is inside of this setter method, if at any point you just type in the variable name `name`, **Java will think** you are **referencing** the **input parameter variable**. So you could imagine that Java would be pretty confused if you wrote the setter method like this:

```
public void setName(String name)
{
    name = name;
}
```

Java would look at this and say something like “`This assignment has no effect`”. And if you think about it, that makes sense, you're essentially saying “*Take what's in the name parameter, and put it back into itself*”. Thus, it would have no effect.

Java isn't smart enough to realize that what you really want to do is: **assign to the instance variable** called `name` (of the `HumanBeing` object) the **value** of the **input parameter** called `name`.

Some other programming languages (like C++) use a convention where they prefix the instance variables with underscores (_). So a setter method using C++ convention would look like this:

```
public void setName(String name)
{
    _name = name;
}
```

This would essentially negate the necessity of using the `this` keyword, as there's no longer a name collision conflict. But I find that convention to be a little annoying when you get into heavy usage of the instance variables. So that's why I say that the usage of the `this` keyword inside of setter methods is mainly for keeping things looking neat and tidy, as you could always just use a different name for your instance variables or your parameter variables to avoid the name collision. But that's entirely up to you. I **would** suggest you lean towards using `this`, as it is a **Java convention**.

Access Modifiers

You have seen Java modifiers in code over and over again, but we've never really talked about them in-depth. So what better time than now, right?

What are Java Modifiers?

Also called **access modifiers**, these are seen in code as `public`, `package`, `protected` and `private`. These keywords exist in Java as a means for encapsulation. These access modifiers will either **allow or deny other Objects** from **being able to execute the code** that is wrapped by the keywords.

What do I mean when I say that it will *allow* or *deny* access? I mean that Java will either let you access whatever Class or method you wish to by allowing the code to compile, or if you're not allowed, your code will simply not compile! So then, if your code won't compile because you don't have the access to another Class or method; then what's stopping you from going into that Class or method and changing the access modifier to be public? Well, nothing is stopping you but this could lead to potential problems in the future.

Programmers restrict access to certain areas of code on purpose, for they don't want other parts of the code to interfere with their Objects. Remember that Objects can interact with each other and change each other's variables. If I was a user in Facebook, I shouldn't have access to change another Facebook user's password right?

Examples of Access Modifiers in Java:

Let's look at some code. I've put together a little program that revolves around houses and neighborhoods. There is a `YourHouse` Object that represents your house (duh). There's a `YourParent's House` Object as well as a `YourNeighboursHouse` Object. I've created a bunch of methods inside of each of these Objects that represents some actions that you may take on a regular day. The example I've outlined is if you lost your keys and are trying to get into someone's house for help.

AccessModifiersProgram.java

```
package com.howtoprogramwithjava.runnable;

import com.howtoprogramwithjava.yourneighbourhood.YourHouse;

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

```

    {
        YourHouse yourHouse = new YourHouse();
        yourHouse.knockOnDoor();
    }
}

```

YourHouse.java

```

package com.howtoprogramwithjava.yourneighbourhood;

import com.howtoprogramwithjava.yourparentsneighbourhood.YourParentsHouse;

public class YourHouse extends YourParentsHouse
{
    protected void enterYourHouse ()
    {
        System.out.println("Entering " + getClass().getSimpleName());
    }

    public void knockOnDoor ()
    {
        System.out.println("You knock on the door of " + getClass().getSimpleName());
        double randomNumber = Math.random();
        if (randomNumber >= 0.5)
        {
            System.out.println("You are greeted at " + getClass().getSimpleName());
            enterYourHouse();
        }
        else
        {
            System.out.println("No one answers");
            goToParentsHouse();
        }
    }

    protected void goToParentsHouse ()
    {
        System.out.println("You are going to Your Parents House");
        if (!this.enterYourParentsHouse())
        {
            YourNeighborsHouse yourNeighborsHouse = new YourNeighborsHouse();
            yourNeighborsHouse.enterYourNeighborsHouse();
        }
    }
}

```

YourParentsHouse.java

```

package com.howtoprogramwithjava.yourparentsneighbourhood;

public class YourParentsHouse
{
    protected boolean enterYourParentsHouse ()

```

```

    {
        if (Math.random() >= 0.5)
        {
            System.out.println("Entering Your Parents House");
            return true;
        }
        else
        {
            System.out.println("No one is home at Your Parents House");
            return false;
        }
    }

    private void changeThermostat ()
    {
        System.out.println("Only your parents can change their thermostat!");
    }
}

```

YourNeighboursHouse.java

```

package com.howtoprogramwithjava.yourneighbourhood;

class YourNeighborsHouse
{
    void enterYourNeighborsHouse ()
    {
        System.out.println("Your neighbor is home, so you enter " +
getClass().getSimpleName());
    }
}

```

Okay, so we've got a lot going on here. One piece of code you may not recognize is the `Math.random()` static method. What's that about? It's a really cool method that will randomly generate a number between 0.0 and 1.0. Why is that helpful? Well, it means I can say that a certain event will happen if the number is between 0.0 and 0.49, and if it's bigger than 0.49, then do something else. This is what I'm doing in my example code above, so if you were to copy/paste the code into your own project, you'll see that the outcome in the console changes almost every time you run the code! Neat!

Anyway, let me list out the important things you should observe in this code:

- `YourHouse` extends `YourParentsHouse`
- `YourNeighboursHouse` doesn't use any modifier keywords
- `YourParentsHouse` has a `private` method

Alright, since `YourHouse` extends `YourParentsHouse` this means that `YourHouse` will have access to all of the `public` and `protected` methods inside `YourParentsHouse`, as well as all of the `public`, `protected` and `private` methods within its own code.

What can we learn from this?

Objects and Methods

public (Objects and Methods)

If a method is declared as `public`, then every Object or method that exists inside of any package in your program will have access to that method. This is equivalent to you leaving the front door of your house WIDE OPEN to the public. You're inviting everyone and anyone in to snoop around 😊

Note: Just because **your Object** is declared as `public` doesn't mean that every **other Object** or method has access to every aspect of **your Object**, for we can still modify the access levels of the **methods within** your Object.

protected (methods only)

If your method (within your Object) is declared as `protected`, this means that the only Objects/methods that will have access to your method are those that extend your Object (or Objects that your Object extends). In other words, any Objects in your Object's inheritance chain, will have access to your Object's `protected` method.

This is seen when `YourHouse` calls the `protected Boolean enterYourParentsHouse()` method. This method is protected, and it belongs to another Object (`YourParentsHouse`), but since `YourHouse` extends `YourParentsHouse`, then we're all good!

package (Objects and methods)

This modifier is not used very often as there aren't many practical applications for it. But for the sake of completion I'll talk about it. This modifier can be observed in the `YourNeighborsHouse` Object. Look at the `void enterYourNeighborsHouse()` method. You'll notice that there is no modifier keyword specified in front of the method declaration. This means that it's a `package` level modifier. That's right, you don't actually put the keyword `package` into your code to specify that it should be a `package` level modifier. Kind of strange, but that's how it's implemented.

So when your method (within your Object) is declared as `package`, this means that only other Objects that belong to your Object's package can see your method. You'll notice that both `YourHouse` and `YourNeighborsHouse` belong to the `package`

`com.howtoprogramwithjava.yourneighbourhood`. So this means that `YourHouse` will be able to access `YourNeighborsHouse`'s package level methods. However, if I were to create a `protected` method inside of `YourNeighborsHouse`, then `YourHouse` would NOT be able to access this method! This is because `YourNeighborsHouse` is NOT inside of the inheritance chain of `YourHouse`. Neat!

private (methods only)

Finally we arrive at the last modifier. When a method is declared as `private`, this means that ONLY the Object that declares the method will have access to it. So this would mean that `YourHouse` will NOT have access to the `private void changeThermostat()` method inside of `YourParentsHouse` (even though they're on the same inheritance chain).

Coding Exercise

So I want you to get a good understanding of what I'm talking about when I say that certain Objects have access to other methods. So I want you to copy/paste my code into your STS IDE program. I want you to navigate to the `AccessModifiersProgram.java` file and at the end of the `main` method, I want you to type in "YourHouse.", and you'll see the code completion popup appear with a list of methods that you can access FROM the `AccessModifiersProgram.java` file. Make note of what methods you CAN see and what methods you CANNOT see. Now play around with the access modifiers for some of the methods inside of `YourHouse` and make some public (be sure to save the code). Now go back and type in "YourHouse." in the `main` method again, and you'll see something different.

Let's summarize

Now that you have an understanding of what Java modifiers are, I'll explain again why they're important. If you were to design an entire web application using Java, and you decided to make everything `public`, you'll quickly see bugs start to emerge as you have other developers work on your code. Using Java modifiers is a good thing and allows for a truly Object Oriented approach to your coding. So remember to design your applications with the real world in mind. If an Object has a method that just wouldn't make sense for any other Object to mess around with, then make it `private`! This is good programming practice, as I've mentioned before, it's called encapsulation – which sounds cool, so it must be good then right?

What is a Constant in Java?

A constant in Java is used to map an exact and unchanging value to a variable name.

Constants are used in programming to make code a bit more robust and human readable. Here's an example:

Imagine you are creating a program that needs to calculate areas and volumes of different shapes, it could look something like this, but this is an example of **WHAT NOT TO DO**:

```
1. public class AreasAndVolumes
2. {
3.     public double volumeOfSphere (double radius)
4.     {
5.         return (4/3) * Math.pow(3.14159 * radius, 3);
6.     }
7.
8.     public double volumeOfCylinder (double radius, double height)
9.     {
10.        return Math.pow(radius * 3.14159, 2) * height;
11.    }
12.
13.    public double areaOfCircle (double radius)
14.    {
15.        return Math.pow(radius * 3.14159, 2);
16.    }
17. }
```

So, this above code will get the job done, but there's something that can be improved. Can you guess how?

We should be using a **constant!** Look how many times we use the `double` value `3.14159`, this value represents pi (π). We should create a constant that will assign the value of π to a variable name. Here's how we do it with some new code:

```
1. public class AreasAndVolumes
2. {
3.     // here we've declared a new variable called PI and assigned
4.     // the value of 3.14159 to this new variable.
5.     private static final double PI = 3.14159;
6.
7.     public double volumeOfSphere (double radius)
8.     {
9.         return (4/3) * Math.pow(PI * radius, 3);
10.    }
11.
12.    public double volumeOfCylinder (double radius, double height)
13.    {
```

```
14.     return Math.pow(radius * PI, 2) * height;
15. }
16.
17. public double areaOfCircle (double radius)
18. {
19.     return Math.pow(radius * PI, 2);
20. }
21. }
```

So now, we have a variable named `PI` declared in the instance variable declaration space. We've done this because we need to use this new constant value throughout our `AreasAndVolumes` class. This constant will function just like any other instance variable with one main exception... we've made the value `final`.

The final Keyword

Here is the magic behind a constant value. When you declare a variable to be `final` we are telling Java that we will NOT allow the variable's "pointer" to the value to be changed.

That last sentence is key to understanding constants, did you read it thoroughly? If not, **re-read it**.

What the `final` keyword means is that once the value has been assigned, it cannot be re-assigned. So if you tried to put in some code later that tries to do this:

```
PI = 3.14159265359
```

You would get a compilation error, and your IDE would tell you that a new value cannot be assigned because the variable has been declared `final`.

Does that make sense? A constant is a constant for a reason, it shouldn't ever change! Now, here's a quick note on conventions in Java. When you name a constant, you should use `UPPER_CASE LETTERS WITH underscores INDICATING SPACES`. Again, it's not mandatory to use upper case letters with underscores to indicate spaces, but it's a convention that programmers use and are familiar with in Java. So why fight it?

private vs public Constants

So what you saw in the examples so far, were just `private` constants. This means that no other Classes in your Java project would be able to use the constant value. Sometimes you want it that way because you like to keep your constants organized into the classes where they'll be used. This is a perfectly acceptable (and encouraged) way to handle your constants, but let's say that you have a constant that needs to be used in multiple Classes. You wouldn't want to declare two separate `private` constants in two Class files right? So I recommend you pick a Class that best fits the constant and declare it as `public`.

Important Note: You may find examples on the internet of people declaring ALL of their constants in ONE file and that file is declared as an interface. I DO NOT recommend this approach, as it is a dated approach to handling constants. Since those times, Java has implemented something called a static import which will negate the usefulness of storing constants in an interface. So let's see an example of a public constant being used with a static import shall we?

Images.java file:

```
1. import java.io.File;
2.
3. public class Images
4. {
5.     public static final String THUMBNAILS_DIRECTORY = "C:\\\\images\\\\thumbnails";
6.
7.     public void doSomethingWithThumbnails ()
8.     {
9.         File thumbnailFile = new File(THUMBNAILS_DIRECTORY);
10.
11.        // do something with thumbnailFile.....
12.    }
13. }
```

AnotherClass.java

```
1. import java.io.File;
2. import static com.howtoprogramwithjava.test.Images.THUMBNAILS_DIRECTORY;
3.
4. public class AnotherClass
5. {
6.     public void doSomethingElseWithThumbnails ()
7.     {
8.         File thumbnailFile = new File(THUMBNAILS_DIRECTORY);
9.
10.        // do something else with thumbnailFile.....
11.    }
12. }
```

One thing to note here is that we've chosen to declare the THUMBNAILS_DIRECTORY in the Images Class, but we made the constant public... this allows us to reference that same value in another Class file, which is what you see in AnotherClass where we use THUMBNAILS_DIRECTORY when creating a File. This was possible because of the static import, like so:

```
import static com.howtoprogramwithjava.test.Images.THUMBNAILS_DIRECTORY;
```

We are importing the constant from the Images into AnotherClass file so that it can be used just like we declared it inside of AnotherClass. Neat stuff!

One other small thing to note here, is the use of the double backslashes when listing the actual thumbnails directory. You'll see that I declared the constant with this value: "C:\\\\images\\\\thumbnails"... this is a bit confusing, because why in the heck am I using two

backslashes (\\\) between directories? Well, this is because in Java, the backslash (\) is a special character known as an escape character. It's used when you want to insert a particular character somewhere without having that character interfere with your code... (not a very good explanation, but here's a great example):

```
String aSentence = "Here's a great quote \"Behind every great man, is a woman rolling  
her eyes - Jim Carrey\".";
```

Notice how I wanted to put the quote ("") symbols inside of my `String` variable, but I wouldn't normally be able to do this, because the quote ("") symbol is used to denote the start and end of a `String`. Well, I get around this by using the backslash (\) escape character.

So, in summary, if you want to actually have Java output a backslash (\) character, you need to escape the escape character (yes I know, it sounds strange). This is done by using two backslashes (\\). In short:

Input: \\

Output: \

Input: \\\\
Output: \\

Make sense?

Good, great, grand!

Chapter 5

Fundamental Concepts



Exceptions

What are Exceptions in Java?

What do they look like in Java code?

When should I choose to use them?

I know at this point you must be yearning to create a fully functional and practical Java program by now. Without further delay, here's the content!

What are Exceptions in Java?

Java has a system that allows you to “handle” exceptional circumstances in code.

What do I mean when I say exceptional circumstances? Well, I mean that when your code is running, things can go wrong. For example, let’s say we specify that we want Java to read some information from a file. We specify the filename and we try to “open” the file. Well, what happens if that file doesn’t exist (perhaps it was moved from the expected location during the time that the program was running). Well, the code will complain! Java will **throw an exception**. If you told Java to “open” a file at a specific location (i.e. “C:\logs\LogFile.txt”) and that file doesn’t exist, what the heck should the code do now?!

Exception Handling in Java

Well, thankfully there’s a way for us to do something in the event of a problem in our code. Here’s an example of how you would handle reading a file in Java:

```
MyProgram.java
package com.howtoprogramwithjava.runnable;

import com.howtoprogramwithjava.fileio.FileIO;

public class MyProgram
{
    public static void main(String[] args)
    {
        FileIO fileIO = new FileIO("C:\\\\aFile.txt");
    }
}
```

FileIO.java

```

package com.howtoprogramwithjava.fileio;

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class FileIO
{
    public FileIO (String filename)
    {
        BufferedReader br;
        try
        {
            br = new BufferedReader(new FileReader(filename));
            String line = "";
            while((line = br.readLine()) != null)
            {
                System.out.println("Reading line: " + line);
            }
        }
        catch (FileNotFoundException e)
        {
            System.out.println("There was an exception! The file was not found!");
        }
        catch (IOException e)
        {
            System.out.println("There was an exception handling the file!");
        }
    }
}

```

Wow! So that's a lot of code. Now, if you were to copy/paste this code into a project that you create (remember to name the files appropriately, and place them into the correct packages), you'll notice that you get the following output when you try to run the program:

There was an exception! The file was not found!

This is because we specified a filename in the constructor of our `FileIO` class and you likely don't have this file on your computer, so when the code tried to find it and it couldn't, it threw an exception!

Now let's try adding this file to your computer, so please create an "aFile.txt" file on your "C:\\" drive and put the following content in it:

This is line 1 of the file

This is line 2

Once you've done this correctly and you re-run this Java program, you should see the following output:

Reading line: This is line 1 of the file

Reading line: This is line 2

If you are able to see that in your console, then you've done it!

More details about Exceptions in Java

Now that you've seen the code for reading a file and the exception handling that goes along with it, let's talk more about the code. The most important thing to note about exception handling is the following:

```
try
{
    // insert code to execute here that may throw an exception
}
catch (Exception e)
{
    // and exception WAS thrown, do something about it here!
}
```

This is called a try/catch block. It's designed so that if you put code between the curly braces {} of the `try` block, then any exceptions that occur will make the code flow jump into the `catch` block. If no exceptions occur, then the **code will flow through the entire `try` block and skip the `catch` block of code.**

This concept of "code flow" is critical to understand, if you want to grasp what exception handling is in Java.

What happens if you don't use a try/catch?

This is a good question, so what **will** happen if there is an exception thrown from a particular line of code, but that line of code is not inside of a try/catch block? It's an ugly error that's what! Let's change our code so that instead of catching the error and just outputting a console message, we **re-throw the exception**.

```
package com.howtoprogramwithjava.runnable;

import java.io.IOException;

import com.howtoprogramwithjava.fileio.FileIO;

public class MyProgram
{
    public static void main(String[] args) throws IOException // we need to add a
throws declaration here
    {
        FileIO fileIO = new FileIO("C:\\\\aFile.txt");
    }
}
```

```
package com.howtoprogramwithjava.fileio;
```

```

import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;

public class FileIO
{
    public FileIO (String filename) throws IOException // we need to add a throws
declaration here
    {
        BufferedReader br;
        try
        {
            br = new BufferedReader(new FileReader(filename));
            String line = "";
            while((line = br.readLine()) != null)
            {
                System.out.println("Reading line: " + line);
            }
        }
        catch (FileNotFoundException e)
        {
            System.out.println("There was an exception! The file was not found!");
            throw e; // this is our new code
        }
        catch (IOException e)
        {
            System.out.println("There was an exception handling the file!");
            throw e; // this is our new code
        }
    }
}

```

What you should notice with the modified code above is that we added `throw e;` inside of the catch blocks. So, let's say that we get a `FileNotFoundException` thrown inside of our `try` block of code. This will cause the code to flow into the first `catch` block and this is because we've specified that the first `catch` block is supposed to handle `FileNotFoundExceptions`.

Note: You'll also notice that we assigned the `FileNotFoundException` to the variable name `e`. This is just a coding convention in Java, so you can name your exceptions whatever you like!

Anyway, like I said, the code will flow into the first `catch` block of code, it will then output our console message stating the file was not found; then it will re-throw the `FileNotFoundException`. Since there is no additional `catch` block to handle the re-thrown exception, we will get a nasty console error and it will look like this:

```

Exception in thread "main" java.io.FileNotFoundException: C:\aFile.txt (The system
cannot find the file specified)
at java.io.FileInputStream.open(Native Method)
at java.io.FileInputStream.<init>(InputStream.java:106)
at java.io.FileInputStream.<init>(InputStream.java:66)
at java.io.FileReader.<init>(FileReader.java:41)
at com.howtoprogramwithjava.fileio.FileIO.<init>(FileIO.java:15)

```

```
at com.howtoprogramwithjava.runnable.MyProgram.main (MyProgram.java:11)
```

This output is called a **stack trace**. It is Java's way to let the programmer know that there's been an exception that was not "handled," and thus the execution is stopped. This output will help the programmer figure out where the exception occurred, thus giving a clue as to how it can be fixed/addressed.

When should I use Exceptions?

Now that you have a better understanding as to what an exception is in Java, the next topic is **when to use exceptions**.

Well, in our example above, we don't actually have a choice. Java makes it mandatory to handle any I/O exceptions. This mandatory handling of exceptions is governed by the `FileReader` class and the `BufferedReader`'s `readLine()` method. If you were to inspect the code of that class and method, you'll notice that both have a `throws` clause on their Class and method declarations:

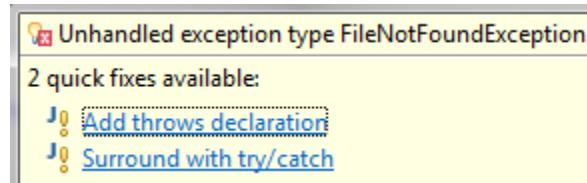
`FileReader.java`

```
public FileReader(String filename) throws FileNotFoundException
{
    super(new FileInputStream(fileName));
}
```

`BufferedReader.java -> readLine method`

```
public String readLine() throws IOException
{
    return readLine(false);
}
```

Because Java has included those `throws` clauses, we are now forced to handle those exceptions. If we **do not** use a `try/catch` block (or specify a `throws` clause on our methods that use this I/O code), we'll get an error in our IDE telling us we need to do something! This error would look something like this:



Now there are other times when it's **not mandatory** that you use exception handling in Java, but it may be beneficial. Examples of this are if you are designing a software application that needs to follow certain business rules. In order to enforce those business rules, you could create your OWN exception class (by creating a new Class and extending the `Exception` Class) and throw THAT exception if a certain condition is met. Let's say, for instance, you are creating a webpage that has a user login feature. When the user tries to login with a password that is incorrect, perhaps you would like to throw an "Invalid User Login Exception?" The design of the system will be up to you or the architect of the system, but this is an option that I've seen used before.

Bonus material

There's one other aspect to the `try/catch` block that I didn't mention, and that's the `finally` block. It looks like this:

```
try
{
    // insert code to execute here that may throw an exception
}
catch (Exception e)
{
    // and exception WAS thrown, do something about it here!
}
finally
{
    // When the code flows out of the try or catch blocks, it
    // will come here and execute everything in the finally block!
}
```

What!? Seriously, there's another aspect to this exception stuff! I know, but it's the last crucial concept you'll need to understand before you can say you understand exceptions in Java. As the code comments state, the `final` block will be executed after the code has either finished executing in the `try` block or in the `catch` block.

So, what's the purpose of the `finally` block? Well, normally it's used to "clean up" after you. An example of this is trying to have your code talk to a database (like MySQL/Oracle/MS SQL Server). Talking to a database is no simple task and it involves steps. The first step is to "open" a connection to the database (very similar to opening a file like we did in this example). Well, what if something goes wrong after we've opened the connection to the database? We need to make sure we "close" the connection to the database or otherwise we'll run into problems (which I won't example here as it's currently not in our scope of learning). The `finally` block allows us to do this "closing" part. Since we can be **confident** that the code will **ALWAYS** flow into the `finally` block after it's gone through either the `try` or the `catch`, then we can throw in our code to "close" the database connection in the `finally` block. Does that make sense?

Note: For the sake of being complete, I'll mention that the finally block doesn't necessarily execute immediately after the code flows past the `try/catch` block, but it will eventually execute the code in the `finally` block... so I wouldn't put any code in the `finally` block that has any dependency on time/point of execution.

Final words on Exceptions

Okay, so I could probably ramble on and on for hours on the topic of exceptions in java, but instead I will stop here and allow this information to sink in a little. The best thing you can do is to use my code examples in your IDE and run the code in debug mode to understand how the code flows

String Manipulation

With this **How to Program with Java**, we will talk about more advanced aspects of String manipulation. With every programming project I have worked on in my career, I've needed to use different strategies related to manipulating Strings. So this topic is critical to learning Java and being a **solid programmer** (in any language really).

What is String Manipulation?

At this point you should be familiar with what a `String` data type is in Java. `Strings` are a popular data type and there are a wealth of tools that can be used to modify their contents. Let's say you have the following code:

```
List<String> websites = new ArrayList<String>();
websites.add("http://howtoprogramwithjava.com/reviews/");
websites.add("http://howtoprogramwithjava.com/programming-101-the-5-basic-concepts-of-
any-programming-language/");
websites.add("http://howtoprogramwithjava.com/consulting/web-application-
development/");
websites.add("http://google.com");
```

Now, let's say we want to isolate two parts of each of these websites, the root URL and the pages within. So, for example, we want to break each one into two parts like so:

1. "howtoprogramwithjava.com", "reviews"
2. "howtoprogramwithjava.com", "programming-101-the-5-basic-concepts-of-any-programming-
language"
3. "howtoprogramwithjava.com", "consulting", "web-application-development"
4. "google.com"

How would we go about accomplishing this? Well, we would need to use a method called `split()`. What this method does is allow you to separate one `String` into an `Array` of `Strings` based on a character of your choosing. In our examples, you'll notice that the forward slash (/) is used to separate the base URL with the individual pages. So this forward slash (/) character would be perfect for splitting up our `Strings`. So, what would that look like?

```
for (String website: websites) // iterate through each website in the ArrayList
{
    System.out.println("website: " + website);
    String[] stringArray = website.split("//");
    System.out.println("Splitting String by // = " + Arrays.toString(stringArray));
    System.out.println("Splitting String further by / ->" +
    Arrays.toString(stringArray[1].split("/")));
```

```
    System.out.println(""); // blank space
}
```

This code will split the `String`s twice. Once with the double forward slash (//) to isolate the “http://” section from the rest. Then it will split the resulting `String` array once again by a single forward slash (/), which will separate all the individual sections of the URL. The output will look like this:

```
website: http://howtoprogramwithjava.com/reviews/
Splitting String by // = [http:, howtoprogramwithjava.com/reviews/]
Splitting String further by / ->[howtoprogramwithjava.com, reviews]

website: http://howtoprogramwithjava.com/programming-101-the-5-basic-concepts-of-any-programming-language/
Splitting String by // = [http:, howtoprogramwithjava.com/programming-101-the-5-basic-concepts-of-any-prog
Splitting String further by / ->[howtoprogramwithjava.com, programming-101-the-5-basic-concepts-of-any-pro

website: http://howtoprogramwithjava.com/consulting/web-application-development/
Splitting String by // = [http:, howtoprogramwithjava.com/consulting/web-application-development/]
Splitting String further by / ->[howtoprogramwithjava.com, consulting, web-application-development]

website: http://google.com/
Splitting String by // = [http:, google.com/]
Splitting String further by / ->[google.com]
```

The `split()` method is very powerful and it allows you to use something called Regex to identify the parts of the `String` to split up. Now, Regex is a fairly advanced topic, but if you feel like learning more about it right now you can check out this [Introduction to Regex](#).

Searching Strings

Perhaps you'll find yourself in a scenario where you'll need to find out if a particular `String` exists INSIDE of another `String`. This will require you to search through your `String`. Let's say we have the sentence “The quick brown Fox jumps over the lazy Dog,” and we want to see if that sentence has the word “Fox” in it. Well, then you would use the `contains()` method. Example:

```
String sentence = "The quick brown Fox jumps over the lazy Dog.";
boolean sentenceContainsFox = sentence.contains("Fox");
System.out.println("Does the sentence contain 'Fox': " + sentenceContainsFox);
```

The output of this will be:

```
Does the sentence contain 'Fox': true
```

One interesting thing to note though is if we made one minor change to the code. What if we change the `contains` method to look for the `String` “fox” (with a lowercase ‘F’). What do you think Java will output?

```
String sentence = "The quick brown Fox jumps over the lazy Dog.";
Boolean sentenceContainsFox = sentence.contains("fox");
System.out.println("Does the sentence contain 'fox': " + sentenceContainsFox);
```

Here's the output:

```
Does the sentence contain "fox?": false
```

Very interesting! This happens because Java is **case sensitive** when it deals with `Strings`. This means that uppercase and lowercase are not treated as equal, **just like when you use variables names when coding!**

The common way to **ignore the case** of your `String` is to send the whole `String` to lowercase, like so:

```
String sentence = "The quick brown Fox jumps over the lazy Dog.";
sentence = sentence.toLowerCase();
Boolean sentenceContainsFox = sentence.contains("fox");
System.out.println("Does the sentence contain 'fox': " + sentenceContainsFox);
```

Matching Strings

Now sometimes we don't need to worry about converting the entire `String` into lowercase to do checks. If you are checking to see if a particular `String` is equivalent to another `String` you could use the following methods:

`equals()`

This allows you to check if two `String`s are exactly equal.

```
String s1 = "fox";
String s2 = "Fox";
s1.equals(s2); // will return false
```

`equalsIgnoreCase()`

This allows you to check if two `String`s are equal to each other while ignoring the case sensitivity.

```
String s1 = "fox";
String s2 = "Fox";
s1.equalsIgnoreCase(s2); // will return true
```

An example of what the `equals()` method comes in handy, is when you need to validate a user's password when they login. You need to ensure that the case matches exactly with the password. Whereas with the username, you don't care as much with respect to case, so you would use the `equalsIgnoreCase()` when validating their username.

Note: Just remember never to compare two `String`s using the '`==`' operator, as this compares the memory address of the actual `String` instead of comparing the actual contents of the `String`.

Substring

Another useful method is the `substring()` method. This allows you to deconstruct a piece of `String` based on an index. What the heck does that mean!? Here's an example to make things clearer:

```
String aString = "This is a sentence that will be deconstructed with the substring  
method";  
String aNewString = null;  
  
// substring takes two parameters, a beginning index and an ending index  
// it will construct a new String based on these indices  
aNewString = aString.substring(8, 45);  
System.out.println(aNewString);
```

This will output:

```
a sentence that will be deconstructed
```

You see what happened there? The “`aNewString`” `String` is made up of the 8th character all the way to the 45th character of the old `String` (“`aString`”).

Try it Yourself

Open your STS IDE and create a `String`. Then type in the variable name that you assigned to your `String` and hit DOT `(.)`, you'll see the list of methods that you can call popup in the auto-complete popup box. In this list you'll see things like `startsWith()` and `replace()`. Read up on what they do and test them out for yourself, it's the best way to learn what you can do with `String` manipulation.



Casting

This is a quick Java tutorial about casting variables. This topic isn't very interesting, but it's definitely something that you should be familiar with if you want to be a programmer (in any language).

What is Casting in Java?

Well, all casting really means is taking an Object of one particular type and "turning it into" another Object type. This process is called **casting** a variable.

This topic is not specific to Java, as many other programming languages support casting of their variable types. But, as with other languages, in Java you cannot cast any variable to any random type.

What are the Rules behind Casting Variables?

It's fairly simple, you remember our talk about how everything in Java is an Object, and any Object you create extends from Object? This was inheritance, and this is important to understand when dealing with casting.

If you are going to cast a variable, you're most likely doing what's known as a downcast. This means that you're taking the Object and casting it into a more "specific" type of Object. Here's an example:

```
Object aSentenceObject = "This is just a regular sentence";
String aSentenceString = (String)aSentenceObject;
```

You see what we've done here? Since the type `Object` is a very broad type for a variable, we are "down casting" the variable to be a `String` type. Now, the question is, is this legal? Well, you can see that the value stored in the `aSentenceObject` is just a plain old `String`, so this cast is perfectly legal. Let's look at the reverse scenario, an "up cast:"

```
String aSentenceString = "This is just another regular sentence";
Object aSentenceObject = (Object)aSentenceString;
```

Here we are taking a variable with a more specific type (`String`) and casting it to a variable type that's more generic (`Object`). This is also legal, and really, it is **always legal to up cast**.

What are the benefits of casting variables?

There are times when you want to get more specific functionality out of a generic `Object`. For example, in my line of work, I deal with web applications, and I'm always dealing with something called a "model." This "model" represents data that I want to display on a webpage, and the model is essentially a generic `Map`. The `Map` stores key/value pairs, where the key is a `String` and the value is usually just the generic `Object` type. So, an example of what would appear in this model would be:

```
"name" -> "Trevor Page"  
"email" -> "trevor@javavideotutorials.net"  
"birthday" -> "07-01-83"
```

Here's what that `Map` would look like in code:

```
Map<String, Object> model = new HashMap<String, Object>();
```

This `Map` is a candidate for casting, and here's how we would deal with the casting:

```
String name = (String)model.get("name");  
String email = (String)model.get("email");  
Date birthday = (Date)model.get("birthday");
```

You see how we did three separate casts there? Since all the objects stored in the map are of type `Object`, this means that they are very generic and could most likely be down casted. Since we know that the "name" is a `String`, and the "email" is a `String`, and the "birthday" is a `Date`, we can do these three down casts safely.

This would then give us more flexibility with those three variables, because now we have an actual `birthday Date` object, so we have access to methods like `getTime()` instead of just the default `Object` methods.

This is quite a valuable approach to storing `Objects` in a `Map`, because if we had created the `Map` with something more specific than `Object` as the value, then we would be constrained to only storing `Objects` of that specific type (and its sub-classes).

What are the downsides to Casting?

Well, there is a certain amount of risk that goes along with down casting your variables. If you were to try to cast something like a `Date` object to an `Integer` object, then you'll get a `ClassCastException`. This is what's known as a run-time exception, as it's really only detectable when your code is running. So, unless you're doing something with error handling, then your program will likely exit or you'll get an ugly error message on your webpage.

So just make sure that if you are doing any down casting, which you're well aware of the type of object you'll be casting.

Summary

To sum up, casting is a very useful mechanism that allows you to write more generic code that will allow you to handle many coding situations. But this mechanism can introduce some risk if you're not careful with what you will be casting.

Loops

I think it's time for us to have an in-depth conversation about Java loops. So, this Java tutorial will be structured around these useful control structures. This topic is yet another one of those that I would classify as "fundamental" when trying to learn how to program with Java.

I've touched on this topic briefly back in my Java tutorial about [control structures](#), so now it's time to dig deeper!

What kinds of Loops are there in Java?

There are three types of loops:

1. For Loops
2. While Loops
3. Do...While Loops

For Loops

In my opinion, the For Loop is the most common of all three types of loops. It is structured around a finite set of repetitions of code. So if you have a particular block of code that you would want to have run over and over again a **specific number of times** the For Loop is your friend. A common use of this loop is when you have a `List` of items that you need to "iterate" over for processing. Let's say you have a `List of Contacts` in your address book, and you want to send an email to all of them. You could iterate over your `List of Contacts` and extract each e-mail address, like so:

```
public class QuickTest {  
  
    public static void main (String[] args)  
    {  
        // get some pre-existing list  
        List<Contact> contactList = getContacts();  
        // find out how big the list is  
        Integer sizeOfList = contactList.size();  
  
        // initialize your TO: field for the email addresses  
        String emailToField = "";  
  
        // iterate through the contacts and extract  
        // the email addresses for the TO: field  
        for (int i=0; i<sizeOfList; i++)  
        {  
            Contact contact = contactList.get(i);  
        }  
    }  
}
```

```

        emailToField = emailToField + contact.getEmailAddress() + ", ";
    }

    // remove the trailing comma and space at the end of the String
    emailToField = emailToField.substring(0, emailToField.length()-2);
    System.out.println(emailToField);
}

private static List<Contact> getContacts()
{
    List<Contact> contacts = new ArrayList<Contact>();

    contacts.add(new Contact("abc@abc.com"));
    contacts.add(new Contact("abc1@abc.com"));
    contacts.add(new Contact("abc2@abc.com"));
    contacts.add(new Contact("abc3@abc.com"));
    contacts.add(new Contact("abc4@abc.com"));
    return contacts;
}

private static class Contact
{
    private String emailAddress;

    public String getEmailAddress() {
        return emailAddress;
    }

    public Contact (String email)
    {
        this.emailAddress = email;
    }
}
}

```

If you were to run this code, you would see the following output:

```
abc@abc.com, abc1@abc.com, abc2@abc.com, abc3@abc.com, abc4@abc.com
```

As you can see here, we now have a `List` of (fake) email addresses that we could copy/paste into an email program in the “To:” field, fill in a subject/body and fire off an email. A similar approach is used in the real world to parse a `List` of email addresses that are stored in a database. All of this made easy by using the For Loop!

While Loop

Another looping strategy is known as the While Loop. The While Loop is good when you don’t want to repeat your code a **specific** number of times, rather, you want to **keep looping** through your code until a **certain condition is met** (or rather, keep looping while a certain condition is true).

A typical case for using a While Loop is when reading a file.

```

BufferedReader input = new BufferedReader(new FileReader(new
File("C:\\\\testFile.txt")));
try
{
    String line = null;
    // Here we use a while loop because we won't
    // know many lines the file has when this
    // program runs.
    while ((line = input.readLine()) != null)
    {
        System.out.println(line);
    }
}
finally
{
    input.close();
}

```

So you see here that we've used a `while` loop, and although it's not entirely apparent, we will be iterating over this loop until the end of the file that we've read in (with the `BufferedReader`).

For simplicity sake, let me show you a more basic version of the `while` loop... now you wouldn't normally use this particular implementation of the `while` loop, as it would just keep running and never end... but this is just an example:

```

while (true)
{
    // do something
}

```

So as you can see above, the structure of the `while` loop is pretty simple. There's the keyword `while` followed by a condition inside round braces `(/*condition*/)`. The code flow will be as follows:

1. Check the `while` loop **condition**, if **false**, **don't execute any code inside while block**
2. **if condition is true, execute code inside while block**
3. **repeat step 1**

So, given this workflow, can you see how the second While Loop example will run forever? If you happen to try and test this out, you'll see that your computer will start to slow down and perhaps you'll hear the CPU fan fire up and make some noise! Don't worry, you can just click the red stop button in your console window to stop your infinite loop... or just close your Spring Source Tool Suite (STS). The key thing to note is that using While Loops is slightly dangerous for your code, as you can get stuck in an infinite loop if you don't give the right condition statement. I actually brought an entire QA environment to its knees once when I was a junior programmer, so don't worry if you do the same, everyone needs to learn somehow right?

Do.. While Loop

This last type of loop isn't used very often, mostly because it does the same thing as a While Loop with **one difference**. Remember our workflow for the While Loop? Well, the Do..While Loop does it just a little bit different:

1. Execute code in the Do..While block
2. Check the Do..While loop condition, if false exit Do..While block
3. If condition in Do..While loop is true, repeat step 1

You see the difference? The Do..While Loop will always execute the code in your block **at least once**. Whereas, the `while` loop could skip the whole block of code if the condition is `false` the first time around. So like I mentioned already, the Do..While Loop isn't used too often, and I'm sure there are situations where it would make sense to use one, but I haven't hit that situation just yet in my coding adventures 😊

Bonus Content

Now that you've seen the different types of loops that exist in Java, how about we get a little fancy with our code? Let's say we have a requirement that says you need to write some code that will tell us if a particular `Contact`'s email address exists in a `List` of `Contacts`. Sure we could write a For Loop or a While Loop that would iterate through a list to find an email address, but wouldn't it be silly if we found the particular email address we were looking for, and then just kept on looking through the rest of the `List`? What happens if the `List` of email addresses had a million `Contacts`, and we had our answer on the 2nd `Contact`? That would be quite the waste of time right? Right! So here's a neat trick:

The break statement

Never fear, there's an app... err... a keyword for that! The `break` keyword allows you to exit a loop whenever you like, so let's implement that requirement for finding an email address:

```
public Boolean emailAddressExists(String emailAddress, List<Contact> contacts)
{
    boolean foundEmailAddress = false;
    for (int i=0; i<contacts.size(); i++)
    {
        if (contacts.get(i).getEmailAddress().equalsIgnoreCase(emailAddress))
        {
            foundEmailAddress = true;
            break;
        }
    }
    return foundEmailAddress;
}
```

So you see in the code above that if we've found the email address, we `break` out of the loop and return the result of our `Boolean` variable. So now, if we find the email address we're looking for the second time around the For Loop, we don't waste any time! Neat!

But wait, there's more!

The `continue` statement

This one is very similar to the `break` keyword. You see, like I already mentioned, the `break` keyword will terminate the loop that you're currently iterating through, but what if we don't want to terminate the loop, but rather, we just want to **go to the next iteration sooner**? Well that's what the `continue` keyword is for. A common scenario here would be if you had a few conditional statements that were being executed in a particular loop and if one of them resulted in something particular, then you would go to the next iteration of your loop. For example, what if we changed our requirement above to say that we want to not only find an email address, but we want to find someone with a particular name as well? If we know that we only want to exit our loop when we find a contact by the name of "Trevor" who has the email address "trevor@javavideotutorials.net," then we could just check all the names FIRST, and if it doesn't match what we want, then why would we also check the email address? Here's what the implementation could look like:

```
public Boolean emailAddressExists(List<Contact> contacts)
{
    Boolean foundContact = false;
    for (int i=0; i<contacts.size(); i++)
    {
        // the exclamation mark at the beginning of this condition
        // means NOT... so this means that we're saying to run the
        // code in this IF block only if the contact's name is NOT
        // equal to 'Trevor'
        if (!contacts.get(i).getName().equals("Trevor"))
        {
            continue;
        }
        if (contacts.get(i).getEmailAddress().equalsIgnoreCase("trevor@javavideotutorials.net"))
        {
            foundContact = true;
            break;
        }
    }
    return foundContact;
}
```

Summary

Great! Now you've been exposed to three types of loops (although, only the For Loop and While Loop are commonly used). You also have an idea of which one to use in certain situations... For Loops should be used when you know exactly how many iterations you'll need, and While Loops should be used when you don't know how many iterations you'll need. And finally, you know how to optimize your loops with

the `break` and `continue` keywords! All that's left is for you to play around with them yourself, so by all means, go nuts!

Java UI

One aspect that I haven't touched on yet is Java UI (User Interface). I left this topic out of all the previous tutorials on purpose, because I think it's one of Java's weak points.

First of all, what is a User Interface?

You may have noticed that, so far, there has been no real "interaction" between you and the code you've been writing. All that happens is that you say "Run as Java Application" and then your code may output something into the console. But then your program ends and that's it! There hasn't been any real interaction with you while the program is running. This is where a User Interface becomes really crucial.

An example of a User Interface is a window (or dialog) that displays a form on your screen. This form has fields for you to fill out, like *username* and *password*. You have the ability to type in your username/password combination; then hit a "submit" button, and then the program will spring to life and verify that your username/password is indeed correct. THIS interaction is made possible by a User Interface.

Why do I think Java UI is weak?

I have two main reasons why I dislike Java UI in general:

1. It's not very pretty
2. It's confusing

I realize these two reasons aren't the most compelling reasons for disliking all things Java UI, but it was compelling enough for me and every one of my University computer science friends to use something else! Plus when it comes down to it, most applications you use on a daily basis are NOT using Java UI... that's because they're either web applications (like Facebook or Kijiji) or just built on some other products (like Visual Studio).

If you would like to see a sample of what this Java UI is all about, here's a tutorial that covers one of the methods for creating a UI. This framework is known as Swing:

<http://cs.nyu.edu/~yap/classes/visual/03s/lect/l7/>

Another UI tool is called AWT or Abstract Window Toolkit and Sun has a pretty comprehensive tutorial:

<http://java.sun.com/developer/onlineTraining/awt/contents.html>

What UI should I use then?

Well here's where it gets a bit tricky. You see I am a web application developer, and therefore I do almost all of my work for web applications. So I am very comfortable using the web as my Graphical User Interface. In order to do this yourself, you'll have to learn how to use webpages as your UI. This is where HTML comes in!

What is HTML?

HTML stands for Hyper Text Markup Language and all it does is define a language that allows you to talk to internet browsers (like Internet Explorer, Google Chrome or Firefox). If you understand "how to program with HTML," then you can create some great user interfaces for your audience... well... so long as you're also good at knowing what a good website looks like. If you're like me, most programmers aren't designers! I may know how to code, but to make a website LOOK good may take me just as long as creating all of the functionality for the website!

HTML is nowhere near as complex or difficult as Java programming and it's MUCH easier to get started with creating your first webpage using HTML than creating your first Java program. So no worries ladies and gentlemen, with my guidance, you could be an HTML rock star in a few weeks!

In any case I don't want to start teaching HTML just yet, as I want to focus on Java.

Operators

So far we've only looked at some fairly simple control structure conditions, by this I mean we've seen things like:

```
if (age < 13)
{
    System.out.println("You are a child.");
}
else if (age < 20)
{
    System.out.println("You are a teenager.");
}
else
{
    System.out.println("You are an adult.");
}
```

This works well if you don't have a very complicated system to work with, but if you're building a real application, then chances are you'll need to make use of more advanced Java Operators!

What is a Java Operator?

A Java operator is the symbol that you put in the conditional statements of your control structures (for the most part). There are other examples of when you could use certain Java operators, but I'll touch on those at the end of this Java tutorial. So, for the example above, the operator we are using is the "less than" operator (<). So then you could probably guess that there is also a "greater than" (>) operator right? Right! You've probably seen me use it in other tutorials, but what else have you seen? Here's a list of some common Java operators:

Equality and Relational Operators

Operator	Description
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

These should seem fairly straight forward to you, so I'll skip right into the conditional Java operator...

Conditional Operators

Operator	Description
&&	Conditional-AND
	Conditional-OR
?:	Ternary (shorthand for if-then-else statement)

Now **these** conditional operators are more interesting, as they allow you to put many operators together... here's an example of how they'd be used:

```
if ((age > 0 && age < 13) || (age > 19))
{
    System.out.println("You are not a teenager");
}
else
{
    System.out.println("You are a teenager");
}
```

Here I used three different relational operators (less than and greater than) and sprinkled in two conditional operators (&& and ||) all in **one** control structure (**if** statement). Now here's a curve ball, you can also throw in the use of the NOT (!) operator. This will reverse the Boolean logic of any statement. If we were to apply a NOT to our example above, we would have to change the System.out statements like so:

```
if (!((age > 0 && age < 13) || (age > 19)))
{
    System.out.println("You are a teenager");
}
else
{
    System.out.println("You are not a teenager");
}
```

You see how I added that NOT (!) symbol in the IF statement? I added yet another bracket that surrounds the ENTIRE **if** condition. The way you would read this is to determine what would happen without the NOT logic first, THEN just reverse your output. So if the **age** variable was 14, I would say, "Is $14 > 0$ and is $14 < 13$ ", "OR is $14 > 19$ "? In this case I would say NO, none of these statements are true, BUT we then need to switch the outcome because we have a NOT (!) operator that wraps our entire logic, so our NO turns into a YES (or true), so the output would be "You are a teenager"... which is true, if you're 14 then you're a teenager.

Be careful when using the NOT (!) operator, as it can sometimes make figuring out the logic quite complicated. Here's a good example of over-use of the NOT (!) operator to over complicate an `if` statement:

```
if (! (age > 0) || !(age < 13) && !(age > 19))
{
    System.out.println("You are a teenager");
}
else
{
    System.out.println("You are not a teenager");
}
```

Believe it or not, this logic is still valid (I know, because I tested it!). But it's so confusing to read, so please, for the sake of all developers out there, avoid using the NOT operator if you can.

Now, there was one more conditional operator in that chart above that I haven't talked about yet, and that's the ternary operator!

Ternary Operator (also a Conditional operator)

This Java operator is another one of those operators that you should only use when it'll result in code that's easy to read. Someone could easily get carried away with using these ternary operators and it just makes the code almost unreadable...

Having said that, let me show you what this ternary Java operator does!

Using the ternary operator is a shortcut for writing an "If true do this, else do that" block of code. So if we go back to our teenager example that we've already seen:

```
if ((age > 0 && age < 13) || (age > 19))
{
    System.out.println("You are not a teenager");
}
else
{
    System.out.println("You are a teenager");
}
```

This code can be re-written using the ternary operator like so:

```
Boolean isTeenager = ((age > 0 && age < 13) || (age > 19)) ? true : false;
```

Now this looks pretty busy, so let me show you a basic version of what this ternary operator looks like:

```
Boolean isAnAdult = (age > 19) ? true : false;
```

So all this ternary operator does is evaluate the condition **before** the **question mark**, and if the condition evaluates to `true` then it will assign the value to the **left** of the colon to our Boolean variable. If the condition evaluates to false, then it'll assign the value to the **right** of the colon to our Boolean variable.

So in our more basic version, we evaluate the condition before the question mark (`age > 19`). So let's say `age = 20`. It would say, "is `20 > 19?`," **yes**, then assign the value on the **left** of the colon to our variable. So what's to the left of the colon? The Boolean value `true`, so this would be assigned to our `isAnAdult` variable. In the reverse case, if `age = 13`, then we evaluate the condition "is `13 > 19?`" **nope**, so then we assign the value to the **right** of the colon, which in this case is false.

Again, this is just a shortcut way to write code, as the `isAnAdult` code could just be written as:

```
Boolean isAnAdult;
if (age > 19)
{
    isAnAdult = true;
}
else
{
    isAnAdult = false;
}
```

So, 9 lines of code versus 1 line of code. Some would say that using ternary is better because you write less code, but, I would say that you should only use the ternary operator if your code is easily readable. If it isn't, then I would favor **MORE** lines of code **to be clearer**.

Arithmetic Operators

Operator	Description
<code>&&</code>	Conditional-AND
<code>+</code>	Additive operator (also used for String concatenation)
<code>-</code>	Subtraction operator
<code>*</code>	Multiplication operator
<code>/</code>	Division operator
<code>%</code>	Remainder operator

These arithmetic Java operators are also pretty straight-forward. You've seen that the `+` operator can be used to add two numbers together OR to concatenate two `Strings` together (in the Strings Java tutorial). Subtraction, division and multiplication are all self-explanatory; these are used for mathematical operations (just like using a calculator!). **But**, one thing you haven't seen is the remainder operator. So what's that all about?

Remainder Operator

This is used to determine what the remainder is in a division operation. For example, what's 15 divided by 7? Well, it's 2 with a remainder of 1. What's 7 divided by 8? It's 0 with a remainder of 7! This is what the remainder operator (%) tells you, just the remainder of a division operation. So what does it look like when used in code?

```
Integer remainder = 833 % 28;
System.out.println("remainder is: " + remainder);
```

This would output **21**. This is pretty easy right? $833 / 28 = 29$ with a remainder of 21. So the `remainder` variable gets the value of 21, and we output it!

Now the question is what's a real world scenario for the use of the remainder operator? Well, I've really only seen it used for displaying rows of a report in a nice way. What I mean by that is you'll have a report printing on the screen and each row of the report will alternate their background colors (between white and grey). This is accomplished by using the remainder operator, sort of like this:

```
Integer numberOfRows = report.getNumberOfRows();

for (int i=0; i<numberOfRows; i++)
{
    String bgColor = (i % 2 == 0) ? "white" : "grey";
    // then we would use this bgColor for the reports output
}
```

So in this example we used the remainder operator inside a ternary operator. What will happen here is that as we go through each of the rows, the variable `i` will increment. Here's what each iteration will look like:

i	(i % 2 == 0)?	bgColor
0	true	white
1	false	grey
2	true	white
3	false	grey
4	true	white

So you see what happens as a result? You will get alternating white and grey background colors, which makes reading reports much easier, as it lines up each of the rows for you 😊 Kind of nifty right? Okay not really, but still, it's a great example for the use of the remainder operator!

Unary Operators

Operator	Description
<code>++</code>	Increment operator; increments a value by 1
<code>--</code>	Decrement operator; decrements a value by 1

You've seen me use these before in my `for` loops. This is yet another shortcut way of writing code, and this is best explained with a code example:

```
int i = 0;
i = i + 1;
// what does i equal now?
// well since i = 0, it's saying i = 0 + 1
// so now i equals 1
i = i + 1;
// what does i equal now?
// well now, since i = 1, it's saying i = 1 + 1
// so now i equals 2
i++;
// what does i equal now?
// well now, since i = 2, it's saying i = 2 + 1
// so now i equals 3
```

You see? Saying `i++` is the same as saying `i = i + 1`. Piece of cake!

Now you might ask, is there a shortcut way of writing something like `i = i + 238`, why yes... yes there is!

```
int i = 0;
i += 238;
// now i = 238, because this is the same thing as saying
i = i + 238;
```

So, all of these shortcuts are the same for subtraction, division and multiplication as they are for addition (with the exception of `**` and `//`). `**` doesn't mean anything, and `//` is used to write in comments as you've seen already. But `i *= 10` is valid, it just means `i = i * 10`. Or `i /= 15` which is the same as `i = i / 15`. You get the idea.

These shortcuts are ones that I would always recommend using, as they don't make the code less readable in my opinion. I've never had a problem understanding what anyone meant when they were used.

For the sake of completion

There are also operators known as "Bitwise and Bit Shift Operators." I don't really want to talk about them, as it would be a somewhat confusing topic for me to cover at this point in your learning, and I don't think you would get much out of it, as these operators are not used very often in the real world. If you DO feel like bonus points and you want to learn about these operators, check out [roseindia's tutorial](#).

Summary

Wow, that's a lot of operators and shortcuts! These Java operators are (for the most part) very commonly used in the Java programming language and I think you'll become very comfortable with them as you program with Java. It's also a very universal topic, so you'll be able to carry this knowledge around in other programming languages if you choose to learn another one. So it's definitely worth learning 😊

Enums

Enums is a neat topic that applies to most programming languages (not just Java)!

What is an Enum?

It's short for Enumerated Type, which just means that Java allows you to define your own special variable type. This becomes helpful when you would like to express something in more human readable terms. For example, let's say you would like to make an application that allows you to play a card game, like poker. Wouldn't it be nice if you could assign numeric values to all of the cards? Like so:

- 2 -> 10 = 2 -> 10
- Jack = 11
- Queen = 12
- King = 13
- Ace = 14

Well, enums make this a piece of cake. Let's create a Card enum!

```
package com.howtoprogramwithjava.business;

public enum CardValue
{
    TWO(2),
    THREE(3),
    FOUR(4),
    FIVE(5),
    SIX(6),
    SEVEN(7),
    EIGHT(8),
    NINE(9),
    TEN(10),
    JACK(11),
    QUEEN(12),
    KING(13),
    ACE(14);

    private int cardValue;

    private CardValue(int value)
    {
        this.cardValue = value;
    }

    public int getCardValue()
    {
        return cardValue;
    }
}
```

Voila! We've now outlined what a `CardValue` enum could look like, so now let's define what the Card's `Suit` could look like:

```
package com.howtoprogramwithjava.business;

public enum Suit
{
    HEARTS,
    SPADES,
    CLUBS,
    DIAMONDS;
}
```

Alrighty, so we have now defined what a `Suit` and `CardValue` looks like as Enums in Java. Things that you should note are that Enums need to have a package or private scoped constructor, if you try to specify anything else, Java will fail the compilation of your code. You can also get away with not defining a constructor at all (like I did with the `Suit` Enum). This is because you aren't meant to instantiate Enums, you're supposed to reference the values in a static way, as they are meant to be constant values.

If you don't quite understand what I mean when I say the values should be referred to in a static way, just hang in there, for I'll have an example soon. First, let's put the `CardValue` and `Suit` together into a `Card` object to pull everything together.

```
package com.howtoprogramwithjava.business;

public class Card
{
    private Suit suit;
    private CardValue cardValue;

    public Card (CardValue cardValue, Suit suit)
    {
        this.cardValue = cardValue;
        this.suit = suit;
    }

    public Suit getSuit()
    {
        return suit;
    }

    public void setSuit(Suit suit)
    {
        this.suit = suit;
    }

    public CardValue getCardValue()
    {
        return cardValue;
    }

    public void setCardValue(CardValue cardValue)
```

```

    {
        this.cardValue = cardValue;
    }
}

```

Okay nothing new going on in that code and it's a standard Object that defines some private (encapsulated) variables that are made publicly visible via getter and setter methods, but you can see how this makes sense right? A `Card` is simply made up of a `CardValue` and a `Suit`! Just like real life! Don't you just love Object Oriented Programming?

Okay, so let's talk a bit more about what I meant when I said that the Enums have to have a package or private level constructor. I said that this is because you shouldn't be able to instantiate the class, for the contained values should be referenced in a static fashion. Let's see what I mean with an example:

```

package com.howtoprogramwithjava.business;

import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;

public class Deck
{
    private ArrayList<Card> deck;

    public Deck ()
    {
        this.deck = new ArrayList<Card>();
        for (int i=0; i<13; i++)
        {
            CardValue value = CardValue.values() [i];

            for (int j=0; j<4; j++)
            {
                Card card = new Card(value, Suit.values() [j]);
                this.deck.add(card);
            }
        }

        Collections.shuffle(deck);

        Iterator<Card> cardIterator = deck.iterator();
        while (cardIterator.hasNext())
        {
            Card aCard = cardIterator.next();
            System.out.println(aCard.getCardValue() + " of " + aCard.getSuit());
        }
    }
}

```

Wow! There is a lot going on here. The basic concept of this code is that we want to properly represent a `Deck` of `Cards`. So, we all know that there are 52 `Cards` in a `Deck` right? This means that we should

iterate through all of our `CardValues` and `Suits` in order to match them up with each other. We do this by referring to the Enums in a static way like so:

```
CardValue.values();
// and
Suit.values();
```

If you hover your mouse over the `values()` part of the code in your IDE, you'll see that when you invoke the `values()` method, you'll get an `Array` that represents every value inside the Enum. So if I were to invoke `Suit.values()`, I would get an array of values back like so:

```
[HEARTS, SPADES, CLUBS, DIAMONDS]
```

It is in that **exact** order. It's important to mention that it's in that exact order because that's the order in which they are defined in the `Suit` object. Java is very particular about the way it does things. It doesn't like to give you back an `Array` in any random order.

So, when you keep that in mind, it helps to understand what I'm doing in this `Deck` code. As you scan through the `Deck` code you'll notice that I have two `for` loops and one is nested inside the other. The outer `for` loop is running from 0 to 12 and the inner loop is running from 0 to 3. So, this means that we'll ensure that we get every single combination of `CardValue` and `Suit`.

Note: I said 0 to 12 even though the `for` loop says `i=0; i<13; i++`, but I'll leave it to you to think things through and try to understand why this is.

So, at this point we'll just have a `Deck` of `Cards` that are neatly in order. So, like any good `Deck` of cards, we want to mix them up so they're random right? Well, lucky for us, we used an `ArrayList` Collection to represent our `Deck` of `Cards`, and inside the Collection class, we have a helper method called `shuffle`.

This method is used to randomize **ANY** `List` Collection. Neat 😊

So, after we've shuffled our `Deck` let's just output what the `Deck` or `Cards` looks like!

```
THREE of SPADES
FIVE of SPADES
SIX of DIAMONDS
QUEEN of SPADES
FIVE of DIAMONDS
THREE of DIAMONDS
SEVEN of CLUBS
```

SIX of SPADES
QUEEN of HEARTS
KING of CLUBS
KING of SPADES
SEVEN of SPADES
KING of HEARTS
THREE of HEARTS
JACK of CLUBS
FOUR of DIAMONDS
TEN of DIAMONDS
TWO of CLUBS
EIGHT of SPADES
FOUR of CLUBS
EIGHT of CLUBS
TEN of SPADES
JACK of SPADES
THREE of CLUBS
NINE of CLUBS
SEVEN of HEARTS
NINE of DIAMONDS
ACE of CLUBS
SEVEN of DIAMONDS
FIVE of CLUBS
TWO of SPADES
TWO of HEARTS
KING of DIAMONDS
EIGHT of DIAMONDS
NINE of SPADES
QUEEN of DIAMONDS
EIGHT of HEARTS
JACK of DIAMONDS
TEN of HEARTS
JACK of HEARTS
TEN of CLUBS
ACE of SPADES
FIVE of HEARTS
ACE of DIAMONDS
TWO of DIAMONDS
FOUR of SPADES

SIX of CLUBS
SIX of HEARTS
FOUR of HEARTS
NINE of HEARTS
ACE of HEARTS
QUEEN of CLUBS

And there you go, a nice neat shuffled Deck. Don't you just love Object Oriented programming? You see how I was able to explain a real world scenario of a Deck of Cards by using the names of the actual types in Java? It makes your code so much more readable and understandable.

Java Practice Assignment #3 – The Anagram

Solution for Assignment 2

First let's start off with the solution for assignment 2 before we jump into our next assignment.

[Click Here to download](#) the source files for assignment 2's solution

For your viewing pleasure, here's the video walk-through for assignment 2 as well:

[Click Here for Video](#)



Assignment #3 – The Anagram

[Click Here to download](#) the assignment files for Java practice assignment #4. Video instructions on how to install these files into your IDE can be [found here](#).

This is one of my favorite types of assignments, it's an algorithm assignment. This type of assignment is designed to test your skills at creating an algorithm that will solve the presented problem. Remember that there are MANY ways to solve this problem; your goal should be to create code that is as efficient as possible. Your task for Java practice assignment #4 is to code an anagram solver. First of all, we'll define the term "anagram" for this assignment:

An anagram is considered to be a pair of words that are made up of the exact same letters. Think of it like taking one word, then just scrambling the letters around until you can spell another word. For the purposes of this assignment we'll only be dealing with single word anagrams, as there are certainly anagrams that can be formed by multiple words (but let's not worry about those ones). Here are some examples of valid anagrams:

care -> race

tool -> loot

cloud -> could

An example of words that are NOT anagrams:

tool -> toll (doesn't have the exact same number of letters)

cloud -> clouds (one word is longer than the other)

So, your task will be to create a method that will return true or false (anagram or NOT an anagram)

based on the two `Strings` that will be passed in. Be sure to follow the instructions included in the

assignment files!

Chapter 6

Advanced Topics

Java Multithreading

What is it?

In Java, a Thread is essentially the Object that represents one piece of work. When you start your application and it starts to run, Java has “spawned” (created) a Thread and this Thread is what will carry out the work that your application is meant to do. What’s interesting to note, is that one Thread can only do one particular task at a time. So that would mean it’s a bit of a bottleneck if your entire application just works off of one Thread right? Right!

Java multithreading allows you to do multiple tasks **at the same time**. This is possible because modern day computers have multiple CPUs (CPUs are the brain of your computer, and it has a bunch!). One CPU can work on one Thread at a time (unless your CPUs have hyper-threading, in which case it can handle two at a time). So this means that if your computer has 4 CPUs with hyper-threading technologies, your code could potentially handle 8 Threads at the same time. Neat!

The implications of this are that you can take your code and make it perform MUCH better by introducing the use of multithreading. That is of course, if your program would benefit from the use of multithreading, some applications are fairly simple and things would just get over-complicated by adding in Thread logic.

How do we use it in Java?

Well, it’s really not too tough to implement the use of Threads in Java. The trick is ensuring that all the Threads cooperate properly with each other... but I’ll get into that after I show you an example of how to set yourself up with Threads.

All you need to do to use Threads is to have an Object implement the Runnable interface and override the run method. Here’s an example of a Class named Worker.

```
1. public class Worker implements Runnable
2. {
3.     public static void main (String[] args)
4.     {
5.         System.out.println("This is currently running on the main thread, " +
6.                             "the id is: " + Thread.currentThread().getId());
7.         Worker worker = new Worker();
8.         Thread thread = new Thread(worker);
9.         thread.start();
10.    }
11.
12.    @Override
13.    public void run()
14.    {
15.        System.out.println("This is currently running on a separate thread, " +
16.                            "the id is: " + Thread.currentThread().getId());
17.
18.    }
19. }
```

When this code is run, here's the output I get on my computer:

```
This is currently running on the main thread, the id is: 1
This is currently running on a separate thread, the id is: 9
```

So you can see here that there are two different `Threads` running here. Now in this example, there's really no need to be using two different `Threads` because the flow of this code is linear. The trick here would be to introduce the need for multiple `Workers` to be running at the same time, and to have a lot of work for these `Workers` to carry out. We can easily create lots of `Workers`, I'll just use a `for` loop to create a handful. But how could we simulate lots of work? Well, we could use the `Thread.sleep()` method; this method pauses the thread for a custom defined period of time. When we pause a `Thread`, this would simulate that `Thread` being busy doing some sort of actual work! Sweet, so let's see what that would look like:

```
1. import java.util.ArrayList;
2. import java.util.Date;
3. import java.util.List;
4.
5. public class Worker implements Runnable
6. {
7.     public boolean running = false;
8.
9.     public Worker ()
10.    {
11.        Thread thread = new Thread(this);
12.        thread.start();
13.    }
14.
15.    public static void main (String[] args) throws InterruptedException
16.    {
17.        List<Worker> workers = new ArrayList<Worker>();
18.
19.        System.out.println("This is currently running on the main thread, " +
20.                           "the id is: " + Thread.currentThread().getId());
21.
22.        Date start = new Date();
23.
24.        // start 5 workers
25.        for (int i=0; i<5; i++)
26.        {
27.            workers.add(new Worker());
28.        }
29.
30.        // We must force the main thread to wait for all the workers
31.        // to finish their work before we check to see how long it
32.        // took to complete
33.        for (Worker worker : workers)
34.        {
35.            while (worker.running)
36.            {
37.                Thread.sleep(100);
38.            }
39.        }
```

```

40.
41.     Date end = new Date();
42.
43.     long difference = end.getTime() - start.getTime();
44.
45.     System.out.println ("This whole process took: " + difference/1000 + " seconds.");
46. }
47.
48. @Override
49. public void run()
50. {
51.     this.running = true;
52.     System.out.println("This is currently running on a separate thread, " +
53.         "the id is: " + Thread.currentThread().getId());
54.
55.     try
56.     {
57.         // this will pause this spawned thread for 5 seconds
58.         // (5000 is the number of milliseconds to pause)
59.         // Also, the Thread.sleep() method throws an InterruptedException
60.         // so we must "handle" this possible exception, that's why I've
61.         // wrapped the sleep() method with a try/catch block
62.         Thread.sleep(5000);
63.     }
64.     catch (InterruptedException e)
65.     {
66.         // As user Bernd points out in the comments section below, you should
67.         // never swallow an InterruptedException.
68.         Thread.currentThread().interrupt();
69.     }
70.     this.running = false;
71. }
72. }
73. </worker></worker>
```

So what's the output of the code above? Here's what it says on my computer:

*This is currently running on the main thread, the id is: 1
 This is currently running on a separate thread, the id is: 9
 This is currently running on a separate thread, the id is: 10
 This is currently running on a separate thread, the id is: 11
 This is currently running on a separate thread, the id is: 12
 This is currently running on a separate thread, the id is: 13
 This whole process took: 5 seconds.*

Cool, so that's what I would expect to see. If you look at the code, each worker "sleeps" for 5 seconds, this simulates 5 seconds of work that they are doing. So since we are using multithreading, firing up 5 different workers to carry out 5 seconds of work each should take 5 seconds total. So what happens if we take out the concept of multithreading? How long would this process take? We it would have to create each Worker one at a time and have each one carry out their work in a linear fashion, so it would take 25 seconds to complete. So technically, by implementing multithreading here, we have increased the speed of our processes by 500%! Not too shabby.

Pitfalls of Java Multithreading

One thing I didn't touch on in this article is what's known as Synchronization, and this is the bane of the multithreading programmer's existence. This is because when you introduce the concept of multithreading, you are opening up the possibility of two Threads accessing/modifying the same Object in memory at the same time. When you try to do this, Java will throw an Exception. The solution to this problem is to synchronize access to your Objects so that no two Threads can trample over each other when trying to access any Object's properties/resources.

Another problem that arises with the use of Java multithreading is race conditions. This is when your code's output is based on a particular sequence of events, but with multithreading this is not a great idea, because you can't always guarantee that a particular set of events will happen in any particular order. Here's a wiki article explaining this problem: http://en.wikipedia.org/wiki/Race_condition.

Singleton Design Pattern

I remember back in my days at university, sitting in lecture while a professor droned on about important concepts in programming. The topic at hand was design patterns, and this terminology meant absolutely nothing to me. Perhaps the professor had explained the importance of the subject and I had glazed over it, but nevertheless I was only half listening while I thought of all the other things I could have been doing at that moment.

So I'll make it my mission to ensure that the same fate does not fall upon you, my faithful reader. **Pay attention**, this is one of many topics that are very important as you make your way to becoming a professional programmer.

What the heck is a Design Pattern?

Well, in programming there's been a history of really bad applications and really good applications. So what separates the bad from the good? Well it's all about identifying common scenarios that come up in code, and being able to provide the tried-and-true BEST solution for any given coding scenario. That's what design patterns are all about. They are essentially a collection of well-defined rules that help you to decide how your code should function when dealing with common coding problems. These design patterns have come from the good, the bad and the ugly code of programmers' past... you're familiar with the term "learn from your mistakes"? That's what design patterns are all about.

So, you see why I asked you to **pay attention**? Creating code based on other people's knowledge is a great way to come out ahead, because why would you want to go through the pain of implementing a solution that's full of design flaws, feel the pain of those flaws, then have to fix them (usually in a time-constrained environment with customers screaming). Nobody likes that, so let's plan ahead shall we?

Enter the Singleton Design Pattern

The singleton design pattern is one of many design patterns, but the singleton is one of the most used and well known. Let's see what Wiki has to say about it:

In software engineering, the singleton pattern is a design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system. The concept is sometimes generalized to systems that operate more efficiently when only one object exists, or that restrict the instantiation to a certain number of objects. The term comes from the mathematical concept of a singleton.

Not bad, a decent definition. If I were to put it in terms of the Highlander film, I would say "There can be only one!" So this means that literally only one instantiation of an Object will be allowed, you simply

should not be able to have two separate instances in your application. Okay, fair enough, seems like a simple enough concept but, how is that useful to us?

Well the most popular use of the singleton design pattern are for Objects that deal with things like database connections or controllers in the MVC pattern. I haven't really talked much about databases or MVC, but the reason these things use the singleton design pattern is because it would be a waste of system resources to create more than one instance of those Objects. When you're dealing with an Object that talks to a database, it's fairly "expensive" to create a connection to a database. When the object gets created, one of the first things it will do is create a connection to the database, but once the connection is open you can use it over and over again. So since there's this "cost" of creating the Object, wouldn't it be nice if we had some way of ensuring we would only create it once? Well yes it would be nice, so let's make it a singleton!

Show me an example!

Okay, let's take a look at what a singleton would look like in Java:

```
1. public class SingletonObject
2. {
3.     private static SingletonObject INSTANCE;
4.
5.     private SingletonObject()
6.     {
7.     }
8.
9.     public static SingletonObject getInstance()
10.    {
11.        if (INSTANCE == null)
12.            INSTANCE = new SingletonObject();
13.        return INSTANCE;
14.    }
15. }
```

So, there are a few important elements to this code. The first element is that we have declared the constructor as `private`. If this is your first time seeing a singleton, you likely glazed over that fact (I know I did the first time seeing it). But this is a very important piece of code. The fact that we're making the constructor private, means that we don't want any other code being able to access the constructor... this means of course, that no one will be able to instantiate this Object (they would get a compilation error). That is after-all, our goal here, to only ever have at most one instance. So when you take away access to the constructor, you take away the ability for a random rogue coder to accidentally instantiate the object.

The second important thing to note is that we have a static reference to an `INSTANCE` variable. This is how we will be able to actually work with the Object. If you think about it, how else would you use an Object that you're not allowed to instantiate? So if I can't write the following:

```
1. SingletonObject myObject = new SingletonObject();
```

then how can I use this stupid Object!? Well, when you have a static INSTANCE variable and a static getter for that INSTANCE, you can access it like this:

```
| 1. SingletonObject.getInstance();
```

So what will happen now is the following:

1. If this is the first time the getInstance method has been called, then the INSTANCE variable will be null, and the SingletonObject will instantiate a new version of itself (it can do this because it and only it is allowed to access its own private constructor)
2. If this is not the first time the getInstance method has been called, then the code will NOT instantiate a new Object, it will simply return the existing one

And voila! We now have an Object that will only ever have at most one of itself in existence. Not too shabby eh? Oh, by the way, make sure you understand this concept as well, because you will be tested to reproduce a singleton in the next practice assignment 😊

Java Practice Assignment #4 – Assembly Line

As always, I'll include my solution to the previous assignment before I give you the requirements for the next assignment. Here's my video walk-through of assignment 3:

[Click Here for Video](#)



[Click here to Download my Solution](#)

Requirements for Java Practice Assignment 4

In this week's Java practice assignment, your job is to implement the simulation of an assembly line that will build Cars. Each Car is made up of several components: tires, seats, engine and frame. Each of these components takes a different amount of time to build on their own. Here's the breakdown on the simulated time each component takes to construct:

- Tire – 2 seconds
- Seats – 3 seconds
- Engine – 7 seconds
- Frame – 5 seconds

With these times, you must implement the code that will simulate the construction of each of these components individually, and then once all the necessary components are built you must put them together to make a car. To build a car, you'll need 4 tires, 5 seats, 1 engine and 1 frame. Here's the catch, the assembly line can only and should only be capable of building 3 Components at any given time. You'll need to implement this in your code.

Once you've completed the assignment and all unit tests are passing, try and fiddle with the order of which the Components are assembled on the line. Is there a particular order that provides the fastest building time for a Car?

Download

[Click here to download the source files](#)

Remember

This assignment deals with multithreading and the singleton design pattern. Be sure to read up on these two articles before you try to complete the assignment:

[Multithreading](#)

[Singleton Design Pattern](#)

If you encounter strange issues, just think in terms of the fact that you have multiple threads running at the SAME TIME doing their thing. Most problems can be solved by realizing this, and understanding exactly what's going on in the code. I would recommend the use of plenty of `System.out.println()` statements to be used as logging so you can get a picture of what exactly is happening at any given moment.

Hint: if you are struggling to make the very last unit test pass, do some research on the [synchronized](#) keyword.

Java Recursion

I'd like to preface this by saying that the need to use Java recursion programming does not come up often. I think in my entire professional career I've used one recursive algorithm, though your mileage may vary of course.

So what is Java recursion? In computer programming it's the process of having a method continually call itself until a defined point of termination.

Let's consider the following recursive code:

```
1. public int myRecursiveMethod ()  
2. {  
3.     int aVariable = myRecursiveMethod();  
4. }
```

Warning: Don't actually try to execute this code, as it will never stop.

What sort of conclusions could you draw from this code snippet?

Well the first thing you should take note of is the name of the method: `myRecursiveMethod`. This is just a random name that I chose to use for this method...nothing special going on there... But, take a look at what we're doing inside the method: we're calling a method named `myRecursiveMethod`. Notice anything special there? **Yes**, that's the same method name!

So what happens?

Recursion!

The method will call itself, and it will execute the code inside, which is to call itself, so it will execute the code inside of that method, which is to call itself, so it will execute that code, which is to call itself... You see what I'm getting at here?

This code is missing a terminating condition, this is why it will run forever. So how about we include a terminating condition?

```
1. public int myRecursiveMethod (int aVariable)  
2. {  
3.     System.out.println(aVariable);  
4.     aVariable--;  
5.     if (aVariable == 0)  
6.         return 0;  
7.     return myRecursiveMethod(aVariable);  
8. }
```

Now with this method, we've introduced a terminating condition. Can you spot it?

When our `int` variable holds the value `0`, then this method will not call itself and instead it will simply exit out of the flow. This can be seen from the `return 0` statement.

So now we're in a position where this method will continually call itself with a decrementing value of `aVariable`. So once `aVariable` hits zero, our recursive method is done!

Can you guess what the output would be if we called this method like so:

```
myRecursiveMethod (10)
```

Think about it, try and follow through the code line by line and see what conclusions you can come to... once you've made a guess, go ahead and create a Class file with a `main` method and throw `myRecursiveMethod` in the mix and call it (you'll need to make the method `static`).

Once you've run the program, if you have NO clue what's going on behind the scenes, then I'd suggest debugging by throwing a breakpoint in where the method begins. Step through the code line by line and see what happens (it will clear things up).

So why use Java Recursion?

There are certain problems that just make sense to solve via Java recursion. This is the case because sometimes, when solving problems recursively, you can really cut down on code with your solutions. For example let's take a look at something called the Fibonacci sequence. Here are the first few numbers of this sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21...

If you've never seen the Fibonacci sequence before, then can you figure out what's going on here?

Well, I'm about to explain it, so if you want to try and figure it out on your own, then stop reading.

Here's how it works:

The 3rd number is the sum of numbers 1 and 2 ($0+1=1$).

The 4th number is the sum of numbers 2 and 3 ($1+1=2$).

The 5th number is the sum of numbers 3 and 4 ($1+2=3$).

The 6th number is the sum of numbers 4 and 5 ($2+3=5$).

etc.

What's neat about this sequence is that when you try to sit down and think up an algorithm to solve this problem, you can't help but think of recursion. The reason for this is because the solution to the next number relies on the past two iterations. Now that's not to say that the **only** way to solve this problem is with Java recursion, but it can make the coding much simpler.

The Big Question

So now you know what the Fibonacci sequence is, but here's the big question: **How do you 'solve' this problem with recursion in Java?**

Let's do it!

There are really only two things any recursive code needs to ensure that it will work properly:

A defined ending point.

A constant progression towards that ending point.

So long as you abide by these two rules, you'll be okay. If you fail to abide by them, you might get caught in an infinite loop and you'll have to manually terminate your program (not the end of the world).

Well then, what's the defined "ending point" for our Fibonacci sequence? Well it will come in the form of the problem we wish to solve. The question would be something like this: "What is the 40th number in the Fibonacci sequence?". So there we have it, that "40th" number is the ending point for our sequence.

Okay, so what's our constant progression towards that point? It will be that we will need to iterate through our recursion 40 times, one by one. That's a measurable way of ensuring that we are moving towards the ultimate goal right?

Now with this in mind, let's think about what the code would look like. The first thing we need to do is think of how the Fibonacci sequence can be represented in terms of an equation. So if the first number plus the second number equals the third... and the second plus the third equals the fourth, then we can describe it like so:

$$F_n = F_{n-1} + F_{n-2}$$

Make sense? So the `n` in this case represents the index of any particular number in the sequence. Now obviously, we can't just plug in the value of 40 for `n` and know what the answer is, because we need to start back at the very beginning and work our way up to `n` to figure it out.

Since we need to work our way from the beginning of the equation, then that means we'll likely need to start there with our coding. So how would our code start then?

```
1. public static void main (String[] args)
2. {
3.     int n1 = 0;
4.     int n2 = 1;
5.     System.out.println("n1=" + n1);
6.     fibonacciSequence(n1, n2);
7. }
8.
9. public static void fibonacciSequence(int n1, int n2)
```

```
10. {
11.     System.out.println("n2=" + n2);
12. }
```

That seems pretty good as a start, but there's no recursion going on here... remember we need to call the `fibonacciSequence` method inside of itself to start Java recursion. The only problem is, if we do this now, it will run forever. Remember the two rules, first we need a clear progression towards an end ($F_n = F_{n-1} + F_{n-2}$) and two we need an end!

So what's our ending going to be? Well, earlier I arbitrarily chose to go to the 40th index of the equation, so let's stick with that.

If we are going to be keeping track of which 'index' we're currently 'at' then let's store it as an instance variable. You could also just pass this into the `fibonacciSequence` method, but I don't like passing parameters around unless it's completely necessary. Oh, and we also need to keep track of our ending point, so let's store that as an instance variable as well.

```
1. private static int index = 0;
2. private static int stoppingPoint = 40;
3.
4. public static void main (String[] args)
5. {
6.     int n1 = 0;
7.     int n2 = 1;
8.     System.out.println("n1=" + n1);
9.     fibonacciSequence(n1, n2);
10. }
11.
12. public static void fibonacciSequence(int n1, int n2)
13. {
14.     System.out.println("n2=" + n2);
15. }
```

Okay, so now we've established the starting point, the ending point, but not the recursion. So let's put that in as well:

```
1. private static int index = 0;
2. private static int stoppingPoint = 40;
3.
4. public static void main (String[] args)
5. {
6.     int n1 = 0;
7.     int n2 = 1;
8.     System.out.println("index: " + index + " -> " + n1);
9.     fibonacciSequence(n1, n2);
10. }
11.
12. public static void fibonacciSequence(int n1, int n2)
13. {
14.     System.out.println("index: " + index + " -> " + n2);
15.
16.     // make sure we have set an ending point so this Java recursion
```

```
17. // doesn't go on forever.  
18. if (index == stoppingPoint)  
19.     return;  
20.  
21. // make sure we increment our index so we make progress  
22. // toward the end.  
23. index++;  
24.  
25. fibonacciSequence(n2, n1+n2);  
26. }
```

And Voila! We have our working algorithm for the Fibonacci sequence.

Java Practice Assignment #5 – Recursive Factorial

Alright ladies and gentlemen, this week's assignment is in. But before we get into it:

[Click Here to download](#) my particular solution for Assignment 4.

There will be many ways to solve this assignment, and I'm not saying that my solution is the best solution, but if you were having trouble getting your assignment to work, then mine might help you out.

Recursive Factorial Assignment

Since you have now learned about [recursion in Java](#), now it's time to put that knowledge to the test! Your job will be to implement a recursive method that will be able to calculate any given number's factorial.

First off, if you don't know what a factorial is, allow me to explain. A factorial is a calculation based on multiplying a given number by its decrementing values until you reach the number 1. The notation for this is a given number following by an exclamation mark. Let's look at some examples, as they will make this explanation make sense:

$$2! = 2 \times 1 = 2$$

$$3! = 3 \times 2 \times 1 = 6$$

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

$$6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$$

Does that make sense to you? All we do is take the original number and then multiply it by the number lower than it by 1... rinse and repeat until we get to 1.

So if you remember the important aspects of a recursive algorithm, you'll remember that you must have an ending point, and a constant progression towards that ending point.

[Click here to Download this Assignment](#)

Explanation of How to Setup this Assignment

Here's a quick video that will walk you through how to setup this assignment on your computer:

[Click Here for Video](#)





Mastering Regular Expressions

What is a Regular Expression?

The term regular expression is used to describe a formula that is used for searching through a String. That's really all regex is meant to do, is search through a String for something and then tell you:

1. If it found it
2. Where it found it

Simple right? So let's take a look at how it goes about doing this.

Regex uses two main Objects to carry out its magic, the `Pattern` class and the `Matcher` class. The `Pattern` class is used to identify **what** it is you're looking for. The `Matcher` class is **how** you actually go about looking for it. So again, `Pattern` is what you're looking for and `Matcher` is how you look for it. So let's talk about an example: let's say I have a simple String "Trevor Page", and I want to see if the occurrence of the String "Trevor" can be found. Obviously when things are this simple, it's easy for our brains to answer the question without the need of a computer, but don't worry, things will get more confusing later on!

If we run this "question" through our `Pattern` and `Matcher` code, we'll get the following output:

Found a match starting at index 0 and ending at index 6.

So let's dissect that output. Obviously it makes sense that the String "Trevor" is found in the String "Trevor Page", but why does it say that it found it at index 0 through 6? Well, you should know why it says it starts at 0 right? Because it's a zero based indexing system (just like everything else in Java), but why did it say it ends at 6? The word "Trevor" is only 6 letters in length, and if we're starting at index 0, then shouldn't it end at index 5?

0->5 = 6 letters
0->6 = 7 letters

Right?

Well you are right, but this regular expression stuff uses a zero based index that is inclusive of the starting character and exclusive of the ending character. This is why we must add one to our ending index. Confusing, but that's just how they designed it. Silly Java people!

So how is Regex different from using the `indexOf()` method?

For those of you who are experienced with Java already, you may be asking yourself this question. We already have the means to search through a String for another matching String. It's called the `indexOf` method, and it seems to be easier to use than regex. Here's your answer: regular expressions will search through the entire String and doesn't stop when it finds the first occurrence, it will keep searching and tell you about **every single match** that occurs (including the start and ending indexes).

Also, mastering regular expressions means you will have to learn about all of the advanced searching features that exist with regex in Java. So how about we start talking about those topics?

Metacharacters – AKA Wildcards

You may or may not be familiar with the concept of wildcards, if you're not, I can tell you exactly what they are. Let's think of our example of the String "Trevor Page", let's say we wanted to search that String to see if there are any occurrences of "Trev", "Trevor" or "Trevor's", how would we do this? Well you'd use a wildcard character, more specifically you'd use an asterisk (*). You would simply look for "Trev*". This would successfully match "Trev Page", "Trevor Page", "Trevor's Page".

The one caveat is that the asterisk will match the type of character that precedes it. So this means that since I have the letter "v" before the asterisk, that means that it will be looking to match a word character (i.e. not a number). If I had a letter and then an asterisk, it would be looking for a repetition of numbers. To truly match **anything**, you would need to use both the dot and asterisk together like so: .*

A good article explaining this can be found at [Java Mex tutorials](#).

This **metacharacter** matching is where the real power of regular expressions come from. You've seen the asterisk (*), but what other special characters (or metacharacters) can we use with regex? Well here's the list:

The metacharacters supported by Java regular expressions are: <{[^-=!|]}?*+>

That's a heck of a lot of characters, so let's look at some examples of how they're used shall we?

One of the more commonly used metacharacters is the square brackets []. These are used to group regular text characters (or numbers) together. The "sets" or "groups" of characters (or numbers) are known as **character classes** (not to be confused with Java classes). Let's say you wish to search a String for the occurrence of a few random letters, you would structure your regular expression like so:

[abc]

This will find matches for the letters: a, b or c. So if you were given the sentence "Hello World!", it would match precisely ZERO occurrences because the letters a, b or c don't exist in that String. If you had the String "How are you today?" and you applied the [abc] regular expression, you WOULD get a match.

Two matches to be exact, this is because in the sentence “How are you today?”, there are two occurrences of the letter “a”. How exciting!

Now if you want to get fancier with this stuff, we could move onto using a more complex regular expression. Try this one on for size:

[bch]at

Any guesses on what matches you would get from this regex? It may not be obvious at first, but this will actually make three matches, and those words are: **bat**, **cat** and **hat**. You see why? The first three letters are encased in those square brackets, so they form a character class where Java matches either the letter: b, c or h. Then it will append the search for a String literal “at”. So what do you get when you combine the letters b, c or h with the String “at”? You get: bat, cat or hat.

Just gripping stuff, really! How about this regex:

[^bch]at

Any guesses on what this would match? If you’re mastering regular expressions, then you might be able to guess. The only difference with this regex than the one before is one symbol, the carrot (^). This is actually just a negating symbol, which means it works just like the exclamation mark in Java conditional statements. You would read this regex as saying: any letters other than b, c or h that have the String “at” attached to them. One example of a matching String here would be the word “rat”. This meets the requirements because the “r” at the beginning of the word “rat” is NOT a b, c or h, and it has the String “at” attached to it. Make sense?

Let’s expand a little bit on this concept.

Regex Ranges

A range in regular expressions is defined by using the hyphen (-). The best way for me to demonstrate this is with an example, consider this regex:

[b-d]at

The introduction of the hyphen (-) between the “b” and “d” characters insinuates a range. This means that all letters between “b” and “d” in the alphabet will be matches (ranges are inclusive of their beginning and ending characters). So this particular regex will match the following words: bat, cat, dat.

Ranges can also be used with numbers, or even with numbers in combination with letters. Let’s say you’re interested in searching through some file names, and you want to see if there are any that have the words “file1”, “file2”, “file3” or “files1”, “files2”, “files3”. What would the regex be to match those names?

```
file[1-3]|files[1-3]
```

Here you see that we've made use of the hyphen to define a range of numbers from 1 to 3 and we've also used the "OR" operator (|) to choose between either the word "file" or the word "files".

Predefined Character Classes

In order to master regular expressions, you'll need to know about predefined character classes. You can think of these things as shortcuts, they'll save you from typing a few extra characters. Let me show you what I mean, let's say you want to determine if a String contains JUST numeric characters, absolutely NO word characters. How would you do this? Well you could do something like:

```
[a-zA-Z]
```

This will search your String to see if there are any occurrences of a word character. If this returns anything, then we can say without doubt that your String is not strictly numeric. But look at all those characters you had to type out to convey this desire... just staggering really, 8 whole characters... what a waste of precious finger dexterity. Now let's use a shortcut:

```
\D
```

There you have it, you just saved yourself from typing out an additional 6 characters. Don't your fingers feel rested? Now, my sarcasm **here** may be well founded, but when you really get to mastering regular expressions, you'll see that these shortcuts do save you a lot of time and effort. There are six different predefined character classes, \d for detecting digits, \s for detecting whitespace and \w for detecting word characters, then you just capitalize each one of those to look for the OPPOSITE. In other words, since \d looks for digits, \D looks for non-digits. Since \s looks for whitespace character, \S looks for non-whitespace characters, and then that goes without saying that since \w looks for word characters, then \W looks for non-word characters.

Marvelous.

Time for You to Start Mastering Regular Expressions

Now there's plenty more to regular expressions, but I feel like your brain has been saturated with enough learning for now. I'll let you peruse some of the tutorials I used to learn all about regular expressions myself (docs.oracle.com/javase/tutorial/essential/regex).

I'll leave you with this problem to try out for yourself. In my home country of Canada, we have something called a social insurance number (SIN), this is just like the American social security number (SSN), but it has a slightly different arrangement of numbers. Here's what a Canadian SIN looks like:

```
123-456-789
```

Can you create a regular expression that will properly verify if a given SIN is in the proper format? To be explicit, the proper format is:

any 3 digits followed by a hyphen, followed by any three digits, followed by a hyphen, followed by any three digits and then NO MORE characters at all. Give it a shot and you'll be on your way to mastering regular expressions!



Sorting Collections in Java

Have you run into a situation where you had a `Collection` of Objects (or an `Array`), and you needed to have them ordered in a certain way?

Chances are that if you've been programming with Java and running through the assignments available on this blog, then you have. So how can this be done? And moreover, how can this be done in a way that's fully customizable?

Comparator/Comparable Interfaces

Enter the `Comparator` and `Comparable` interfaces... these are what we use to accomplish the sorting of `Collections` and `Arrays` in a customizable way. You might ask "Why are there two different interfaces used for sorting?". That's a great question, and I'll answer it soon.

So let's start off with a simple example so that you have a clear understanding of what I'm talking about. Let's assume you're presented with the following `ArrayList`:

```
[I, , L, O, V, E, , J, A, V, A, , P, R, O, G, R, A, M, S]
```

With this `ArrayList`, let's say you're given the requirements to sort all of the letters in a unique way.

1. You first need to show all of the vowels first (in alphabetical order)
2. Then then you need to list all the consonants after the vowels (also in alphabetical order)

The resulting `ArrayList` should look like this:

```
[A, A, A, E, I, O, O, , , , G, J, L, M, P, R, R, S, V, V]
```

How Would You Solve this Problem?

Well, if you just tried to use the standard `Collections.sort(theArrayList)`, it wouldn't get you very far. The static `sort` method of the `Collections` Class that takes a `List` will just arrange the letters alphabetically and it won't put all the vowels first.

So you'll need to use a customized sorting algorithm, which means picking between either the `Comparable` or `Comparator` interfaces. So how in the world do you pick between them?

Well, just keep in mind that they both do the same thing in the end. They will both allow you to define a custom sorting algorithm for your Objects.

Comparator

- Use this interface if you want to custom sort on an Object type that you didn't create (i.e. Character, Integer, String)
- This interface is used to compare two different instances of an Object
- It uses the `compare(Object object1, Object object2)` method to perform its magic

Comparable

- Use this interface if you want to custom sort an Object that you have created (i.e. Users, HumanBeing, Vehicle, Caretc.)
- This interface is used to compare the current instance of your Object to another one (i.e. Compare `this` instance of HumanBeing to another HumanBeing)
- It uses the `compareTo(Object anotherObjectOfSameType)` method to perform its magic

Let's Solve our Problem

Okay, so **given the information above**, and knowing that we want to sort an `ArrayList` **do you know which one of the interfaces you would use?**

Seriously, I want you to figure it out. Don't read anything else until you have come up with a guess in your mind.

No cheating...

Well, since we are trying to sort an `ArrayList` of `Strings` then we will need to use the `Comparator`. So let's have a look at the code and how this can be made possible, **here's the example of how to do a regular old sort** of an `ArrayList`:

```
1. public class SortingExample
2. {
3.     public static void main(String[] args)
4.     {
5.         // instantiate ArrayList
6.         List<String> theArrayList = new ArrayList<String>();
7.
8.         // construct the ArrayList with our letters
9.         theArrayList.add("I");theArrayList.add(" ");
10.        theArrayList.add("L");theArrayList.add("O");theArrayList.add("V");theArrayList.add(
"E");theArrayList.add(" ");
```

```

11.     theArrayList.add("J");theArrayList.add("A");theArrayList.add("V");theArrayList.add(
12.         "A");theArrayList.add(" ");
13.     theArrayList.add("P");theArrayList.add("R");theArrayList.add("O");theArrayList.add(
14.         "G");
15.     theArrayList.add("R");theArrayList.add("A");theArrayList.add("M");theArrayList.add(
16.         "S");
17. }
18. </string></string>

```

So you see here that we're populating an `ArrayList` with our "test" sentence and then we invoke the `Collections.sort()` method. The resulting `ArrayList` would look like this:

```
[ , , , A, A, A, E, G, I, J, L, M, O, O, P, R, R, S, V, V]
```

As you can see, that's not what we're looking for; we want all the vowels to appear first. So let's get on with the good code where we make use of our `Comparator`. If you study the `Collections.sort()` methods, you'll see that there's one that takes two parameters, a `List` and a `Comparator`, so this is the method we're interested in using.

I'm going to be making use of something called an Anonymous Inner Type. It's kind of like creating a whole class that can be used within an existing method. It has a special notation, so if you don't quite get it, no worries:

```

1. Collections.sort(theArrayList, new Comparator<String>()
2. {
3.     @Override
4.     public int compare(String o1, String o2)
5.     {
6.         if (isVowel(o1) && !isVowel(o2))
7.         {
8.             return -1;
9.         }
10.        else if (!isVowel(o1) && isVowel(o2))
11.        {
12.            return 1;
13.        }
14.
15.        return o1.toLowerCase().compareTo(o2.toLowerCase());
16.    }
17.
18. /**
19. * This method determine if the String being passed in is a vowel or not.
20. * @param o1 the String that may or may not be a vowel
21. * @return true if the letter is a vowel, false otherwise
22. */
23. private boolean isVowel(String o1)
24. {
25.     return o1.equalsIgnoreCase("a") || o1.equalsIgnoreCase("e") || o1.equalsIgnoreCase(
26.         "i")
27.         || o1.equalsIgnoreCase("o") || o1.equalsIgnoreCase("u");
28. });

```

| 29. </string>

Alright, so you see here that we are using a `Comparator` inside of the `Collections.sort()` method. But we're doing something neat, we're declaring an anonymous inner type for this `Comparator`. What I'm doing here is I'm avoiding having to create a whole new Object, then have that Object implement the `Comparator` interface, then `@Override` the appropriate `compare(String o1, String o2)` method, then instantiate that Object and pass it into the `Collections.sort()` method.

I have avoided all of this pain and suffering by using this little anonymous inner type trick. You'll notice that when I instantiate a new `Comparator` via `new Comparator()` I also immediately insert an open curly brace `{}`. This tells Java that I'm creating an anonymous inner type. If you stop to think about it, you're not allowed to instantiate an interface right? Right! But you can use this trick to get around all that pain I described above.

So! Having explained that, let's talk about that `compare()` method.

Comparator's `compare (Object o1, Object o2)` Method

This method is where the real magic happens in the sorting process. It's a bit confusing, so try and stick with me on this. When sorting through anything, Java will constantly compare one object to another object (often more than 1 time per Object).

Here's a quick video explanation of one sorting process called the "Bubble sort":

So as you heard in the video above, the `compare(Object o1, Object o2)` method is the key to doing the sorting, and it works by returning an int. Now as I said in the video, this method returns one of three values:

negative number – if first argument is less than the second

0 – first argument is equal to the second

positive number – if first argument is greater than the second

Based on these return values, Java will be able to properly sort your Collection. It will be able to do this by "swapping" the values around appropriately. If you're interested in learning more about the specific algorithm Java uses when you invoke `Collections.sort()`, then read up on [Merge Sort](#).

Okay, so that about wraps up this post... as always, if you really enjoy my teaching style and want to get on the fast track for learning Java, head over to [Java Video Tutorials](#) and read up about my fantastic video course. The students currently enrolled in the course have had nothing but great things to say about it, and I'm so excited to be making a real difference in their journey to become programmers.

Java Practice Assignment #6 – Texas Holdem Poker

Before we launch into this assignment, let's take a look at the solution for [assignment 5](#):

[Click Here for Video](#)



Assignment #6 – Fun with Enums

In this assignment we are going to use our knowledge of Enums to create a Texas holdem poker game!

If you aren't familiar with Texas holdem poker, you can become acquainted with the generic [rules of poker here](#).

This particular assignment won't be to implement the entire breadth of the game, but rather, to implement the key elements of the game of poker so that we can expand on it later. So let's talk about the elements that you'll need to implement for this assignment.

You are required to create a Deck of Cards that can be used in our poker game. The deck should be a standard deck of cards consisting of 52 cards made up of 4 different Suits (diamonds, clubs, hearts, spades) with Values ranging from TWO to ACE.

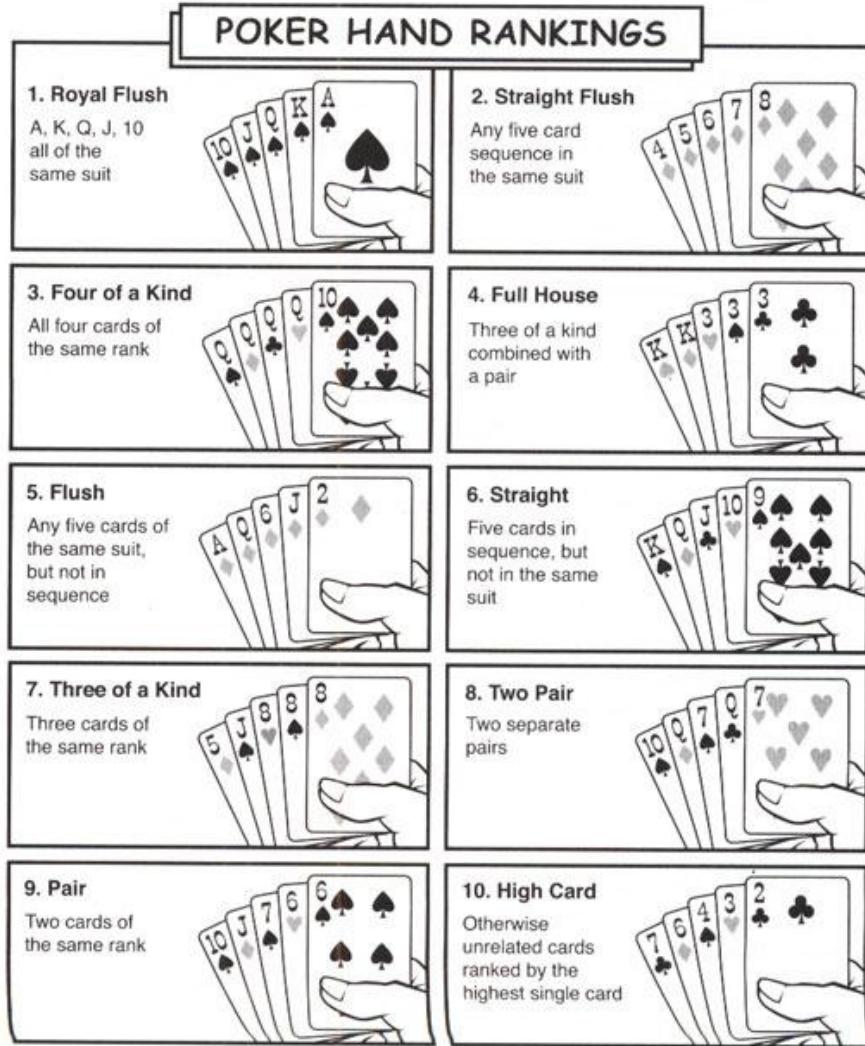
Once you have your Deck of cards setup, the next thing you need to do is be able to:

- Shuffle your deck
- Deal cards to Players

Having said that, you will also need to create Players, these players will have one Hand of cards. Each hand will consist of two distinct cards from the deck.

Not only do you need to deal cards to players, but you also need to deal the "community cards". For this iteration of the assignment, let's just deal five community cards right away. I don't want to put the "betting" aspect into this assignment just yet.

Once you have dealt all players their hands (two cards each), and you have dealt the five community cards, then you'll need to figure out which player has the best hand. To do this, you'll need to know which type of hand beats another, so here's the list:



(picture credit goes to www.betshoot.com)

You'll need to output each players' hand in the console window, as well as the community cards. Once you determine which player is the winner, you should output which player won and with what hand. In the event that there is a tie (i.e. two players hold the same winning hand), you'll need to determine which of the winning hands has the higher value cards. So if both players have three of a kind, then the player with the three higher value cards wins (i.e. Player 1 has three jacks, but Player 2 has three aces... aces > jacks therefore Player 2 wins).

A draw will occur only if the values of the winning hands are exactly the same (i.e. two players have three of a kind with kings and identical 'non-winning' cards)... In the event that two players each have the same winning cards, but different non-winning cards (i.e. Player1 has three aces a king and a jack, Player2 has three aces, a jack and a four), then the player with the higher "kicker" card wins (i.e. Player1 wins because their king kicker beats Player2's jack kicker).

Note: This assignment is more difficult than most other assignments I have posted on this blog. It took me roughly 12 hours of coding to complete this assignment. The good news for you is that I have included a good chunk of the coding in the assignment source files that you can download below.

What You'll Learn

This assignment should make heavy use of [Data Structures](#), namely [Lists](#) and [Maps](#). Everybody's solution will be different, so if you don't use these data structures, then no worries... However, I do believe you will need to use them to make your life a little easier.

You will also learn how to [Sort Collections](#), this will be mandatory, as it will be very difficult to determine which player has won without being able to sort the cards.

My solution even uses a little [Recursion](#), now whether or not you choose to do this is up to you, but I believe it made my solution simpler.

[Click Here to Download the Assignment Files](#)

Explanation of How to Setup this Assignment

NOTE: This video is from Assignment #5, but the same concepts will apply for setting up THIS assignment:

[Click Here for Video](#)



Storage and Database

Java Serialization

The topic of Java Serialization is an interesting one, and can almost be thought of as a stepping stone for learning the concepts of a database.

What is Java Serialization?

Well, it's really as simple as storing an Object (either in memory or in some sort of file or database) in a 'simple to read' fashion. Now when I say that it's easy to read, I mean that it's easy for a **computer** to read the Object, not you or me. What actually happens is that Java will look at your Object, which could have a complicated structure, and turn it into a big ordered stream of bytes.

Okay, I know, it's boring and complicated talk, so let's see what I can do to make it easy to understand. First, let's say we want to serialize a `User` Object. You've seen examples of a `User` Object before, they generally have a username and a password right? So, let's say we have a `User` declared and populated **with** a username and password. Now we want to store that in a file so we can turn off our Java application, and then boot it up again later and retrieve that `User`. This is how serialization is helpful.

Picture this...

Without Serialization

1. You start your Java application in your IDE (via right-click -> Run As ->Java Application)
2. You create a `User` Object
3. You populate that `User` with a username and password
4. Your Java application ends

What happens to that `User` Object you so tediously created and populated with values? Well, **it's GONE!** Once your Java application ends, it frees up all the memory that was in use and POOF that includes your `User`. Well, that sucks. You probably wanted to keep that `User` information

With Serialization

1. You start your Java application in your IDE (via right-click -> Run As ->Java Application)
2. You create a `User` Object
3. You populate that `User` with a username and password
4. **You write your `User` Object to a file using Serialization**
5. Your Java application ends

What happens to that `User` Object now? Well, it's not in memory, because Java has cleaned that up when the application ended. BUT! It now exists inside of a file that is stored on your actual computer's hard drive! Holy crap! That's neat! Now if you write a little more code, you can actually **read** that file and re-create that `User` Object when you fire up your Java application again.

Powerful stuff people

The implication of this topic means that you can now magically store all of the information that you previously had in your Java application while it was running. This is exactly the same kind of basic functionality that you would get from a database, which opens up a lot of doors with what kinds of applications you can create with Java. Now, I will say that using an actual database makes a lot more sense when you start getting into applications that you would create and sell to people, but if you just feel like creating an application that will only be used by yourself or a select few people, then this may be just the ticket 😊

Let's see some code. We want to make our `User` Object serializable, and in order to do that, we just need our `User` Object to implement the `Serializable` interface. Do you still remember what that means? An interface deals with inheritance, which I already talked about. So that would look something like so:

```
import java.io.Serializable;

public class User implements Serializable
{
    private static final long serialVersionUID = 4415605180317327265L;

    private String username;
    private String password;

    public String getUsername()
    {
        return username;
    }

    public void setUsername(String username)
    {
        this.username = username;
    }

    public String getPassword()
    {
        return password;
    }

    public void setPassword(String password)
    {
        this.password = password;
    }
}
```

Okay, so we've seen this kind of code before, except for two things. We've implemented `Serializable` (which is an import from `java.io.Serializable`), okay that seems pretty straight forward. But you may also see that there's a line that says:

```
private static final long serialVersionUID = 4415605180317327265L;
```

What's that all about!? Well, this is used for version control purposes. First of all, if you're using an IDE, then you will most likely be able to have the IDE generate this line of code for you. Seriously, you won't even have to do anything more than hover over the name of the class (in this case `User`) and click on the "Add Generated serial version ID" or something to that effect. Okay great, but what does it mean!?

Well like I said it's used for version control. Here's an example that will help shed some light. Consider this:

1. Create `User` Object
2. Populate it with a username/password
3. Serialize the Object and store it in a file
4. Stop your Java application
5. Change the `User` Object code, and remove the password property and instead add a first and last name
6. When you make a change to the contents of an Object, you need to re-generate your `serialVersionUID`, so delete the existing one, and re-generate the code using your IDE
7. Start up your Java application
8. Try to read your stored `User` Object from the file from step 3
9. You'll get an error stating `InvalidClassException`

You get this error because the `serialVersionUID` of the Object that was stored in the file is now **different** from the one you're trying to read it into. Remember, you removed the password and added first/last name to your new `User` Object. So the Object stored in the file had a password, this new Object doesn't. Since this is the case, Java will automatically throw the `InvalidClassException` exception, and you'll need to decide what to do to handle the situation. This topic of version control/safety is a fairly complex one that has to do with databases and storage mechanisms, so I won't go into any more detail.

Full Example

Alright, now that you know how to enable the `User` Object to be stored (serialized), let's see **how** we actually store this thing!

```
package com.howtoprogramwithjava.runnable;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutput;
```

```

import java.io.ObjectOutputStream;
import java.io.OutputStream;

import com.howtoprogramwithjava.business.User;

public class MyProgram
{
    public static void main(String[] args) throws IOException
    {
        User user = new User();
        user.setUsername("tpage");
        user.setPassword("password123");

        File file = new File("C:\\\\testFile.txt");
        OutputStream fileOutputStream = new FileOutputStream(file);
        ObjectOutputStream outputStream = new ObjectOutputStream(fileOutputStream);
        outputStream.writeObject(user);
        System.out.println("I've stored the User object into the file: " +
file.getName());
    }
}

```

Voila, now I can look on my hard drive and see that a file has been created at C:\\testFile.txt!

Note: I had to use two backslashes (\\\\) when listing my file location, because the backslash (\\) is used as something called an escape character. Its most commonly used when you want to put an actual quote (") symbol in a `String`. For more info on the backslash escape character see [this explanation](#).

Now it's your turn

I have a challenge for you! I want you to now write some code that will take the file on your hard drive, read the file, and re-create the `User` Object you previously serialized. Here's a hint, you'll need to use an `ObjectInputStream`.

Connecting to a Database

So, storing and loading things from files (using [Serialization](#)) is great, but what if you want to use something more robust? This is when you'll need to put things into full throttle and use a real database!

Now this topic can seem scary to those of you who have never dealt with a database before, but when you break the topic down into its fundamental concepts, it is a lot easier to follow along. Let's first try to go over these fundamental concepts before we jump into any sort of coding.

What is a Database?

As I mentioned in the Serialization tutorial, Java applications have a certain lifespan. They only live as long as they are running. Once the application is shut down, all of the Objects that existed in their particular "state" go away. These Objects were all stored in the scope of the Java application's memory, so when the application stops, all is lost.

This would essentially be catastrophic in the case of today's web applications. Can you imagine if you had to re-create your profile and re-import all your pictures on Facebook every time they rebooted their servers!?

This is why databases exist. A database's purpose is to take all of that "state", all of the information that's stored inside of your Objects and variables, and put them in a Database that's (hopefully) neatly organized. So this will mean that as soon as you do upload a picture or post a comment on Facebook, it will immediately get stored in a database so that if they **do** reboot their application, you won't lose everything!

All a database really does is **store data** in a particular **location** (and a particular way) so that the data can be **retrieved later** in a (hopefully) **easy and efficient manner**.

Now, I don't want to start talking about all the different kinds of databases that exist, because that would take up way too many pages and it would probably bore you to death. If you really want to learn everything there is to know about the history of the database and why we use them, I'll let you do some self-guided research on the subject. However, if you do have any specific questions, I invite you to email me at trevor@javavideotutorials.net.

So, now that you understand that a database is used to store information for later retrieval, you may be wondering **how** it stores the information. I keep saying that the information is stored neatly and efficiently, but what do I mean when I say that? Well, that's the job of the **database management system**.

What is a Database Management System?

Here's what wiki says:

The term database system implies that the data is managed to some level of quality (measured in terms of accuracy, availability, usability, and resilience) and this in turn often implies the use of a general-purpose database management system (DBMS). A general-purpose DBMS is typically a complex software system that meets many usage requirements to properly Maintain its databases which are often large and complex. The utilization of databases is now so widespread that virtually every technology and product relies on databases and DBMSs for its development and commercialization, or even may have DBMS software embedded in it.

So, to summarize this definition, a database management system, is the actual system that implements the rules as to how it will go about storing and retrieving the information that you want to save for later use.

The one that I will be using in my examples is known as a Relational Database Management System. It's certainly a mouthful, but once again, it's just a complicated set of words that define something pretty simple.

The Relational Database Management System

The reason why I'm talking about the RDBMS (relational database management system) is because it goes along quite nicely with an Object Oriented language like Java. The RDBMS "describes" its data by using the relationships between the data. For example, if we have a `User` object and a `User` has an `Address` object associated with it, the RDBMS is able to define this relationship and store the information accordingly. This means that inside the database, there would be a section which describes the `User` object, and then a section which describes the `Address` object **AND how they relate to each other**.

The RDBMS' ability to show (or describe) relationships between data is what makes it a lot easier to retrieve the data later. For example, this is handy when we want to retrieve all the data associated with a particular `User` who is currently logging in. When everything is organized and stored with relationships, it's a piece of cake to grab all the data we need.

What Database Operations can I Perform in Java?

Before I dive into the code, I want to give you a high level overview of what you can do with a database. Now, as I've said before, a database's main purpose is to store data for later retrieval, so this probably means we should be able to both store, and retrieve, right? Right.

Insert

This is the operation that will allow you to put/store something into the database. When you insert something into the database, you are creating something new inside of the database.

Select

This is the operation that you will use to retrieve data from the database. You can "select" certain pieces (or all pieces) of data to be loaded into your application.

Update

This is the operation you would use if you wanted to change some existing data in the database. So let's say a User updates their address, you wouldn't want to delete everything and start from scratch, you just want to update the existing data.

Delete

This is the operation you would use when you want to actually permanently remove data from the database. You can target it to delete just one piece of data, or ALL data.

Let's see some code

Okay, now that you're armed with some information about databases, let's see how we could connect one into our code. I'll go back to the example of a `User` trying to login to a system. In the previous example of this, we just hard coded some usernames and passwords, but this time, let's be more practical shall we? Let's actually try to look into a database to retrieve some `Users` that are already stored.

Step 1: Add database library to the classpath

Before we can get started with code, we need to make sure that we have the appropriate library files on the classpath of our application. This is needed in order for Java to understand how to properly "talk" with the database. In this example, we'll be using the MySQL database, so you can grab a copy of their library (JAR) file via this link:

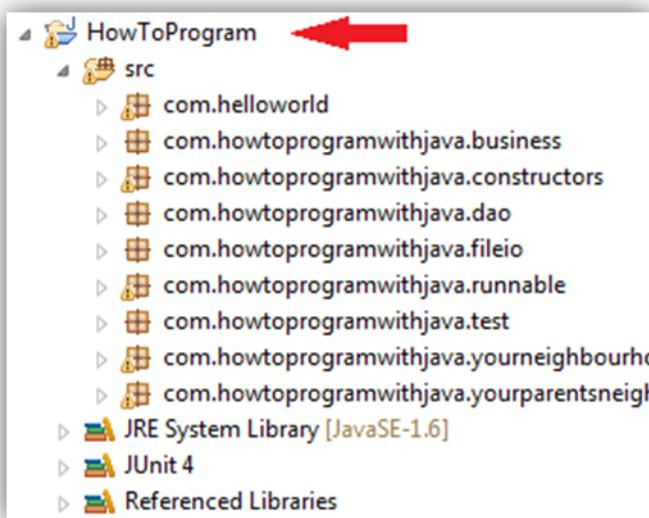
<http://www.mysql.com/downloads/connector/j/>

- When this eBook was published, the MySQL Connector version 5.1.22 was available, so just choose to download the ZIP version.
- When I clicked to download the file, it asked me to login, but there's a link that says "**No thanks, just start my download!**", this will get the file into your hands ASAP.

Once you've downloaded the ZIP file, open it up (I use [7-Zip](#)) extract the JAR file to a directory of your choosing (I prefer to keep it with my actual Java application files, but it's up to you). For the record, the JAR file you should be looking for should look something like this: mysql-connector-java-5.1.22-bin.jar.

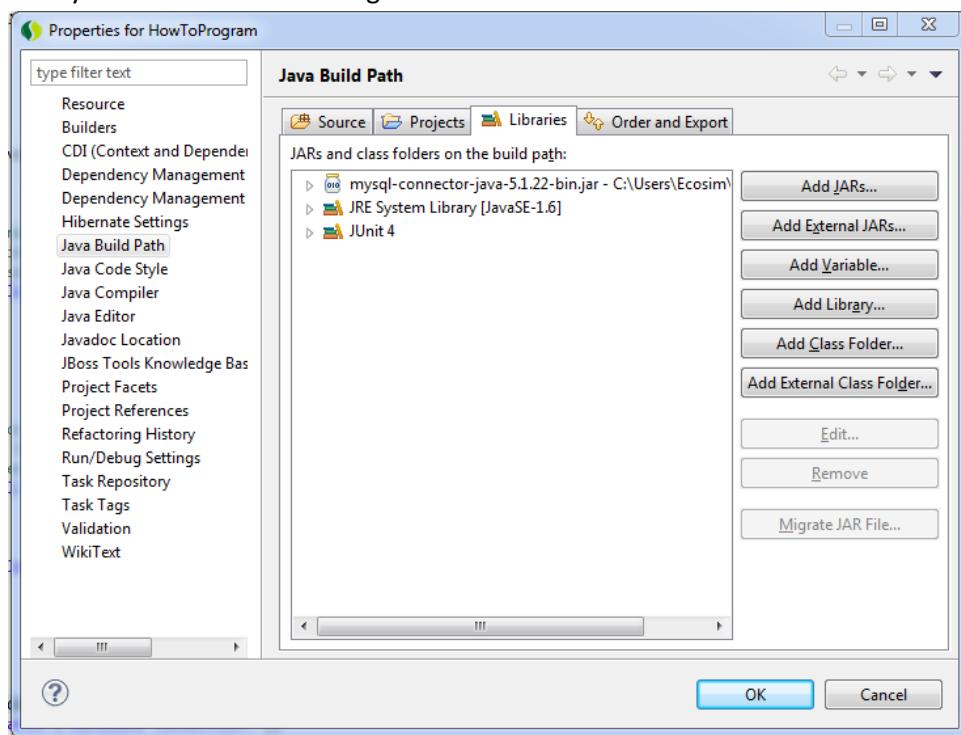
Okay, we're nearly there, we HAVE the file we need, now we just need to tell our Java application where to find it. This will mean we need to add this JAR file to the application's classpath. To do that, you must:

- Go to your STS (or Eclipse), **right-click** on the **project** that you've been working in (in the screenshot below, it's the "HowToProgram" folder). Here's what my **project** structure looks like:



- Choose **properties**
- On the left hand side of the window that appears, click **Java Build Path**.
- In the **Libraries** tab, click **Add External JARs**.
- Choose the **JAR** file that you downloaded and extracted.

- Now you should see something like this:



Excellent, if you've made it this far, then you're in great shape. That was the annoying setup part that's associated with databases, you should really only have to do that whole process once.

Step 2: Download and Install the MySQL RDBMS

This process is a little more involved than just grabbing a file and unzipping it. So I've created a video tutorial that will help you through this step of the process. Just follow this link:

<http://youtu.be/DQULyZAuJww>

Step 3: Write the Code!

Okay, so this is the part you've all been waiting for. Let's dive into some code. As I mentioned before, we'll be working with the concept of `Users` stored in a database that we need to retrieve in our Java application. So in order to accomplish this task, we need to create an Object that can be used to fetch the data from our database. This file is known as a DAO, a Data Access Object. It's just a regular Java Object that you're very familiar with, it's just we're calling it a DAO, as it's a common practice in programming to use this file name. Let's see our UserDao!

```
package com.howtoprogramwithjava.dao;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
```

```

import java.util.List;

import com.howtoprogramwithjava.business.User;

public class UserDao
{
    // this is the URL that is needed to connect to your
    // database. Here we're using MYSQL, a free RDBMS.
    // The database name I've used is 'test'
    String url = "jdbc:mysql://localhost:3306/test";
    Connection con = null;

    public UserDao ()
    {
        try
        {
            // this is the driver for the MySQL RDBMS,
            // each database system has a different class
            // that is used to connect to the DBMS.
            Class.forName("com.mysql.jdbc.Driver");
        }
        catch( Exception e )
        {
            System.out.println("Failed to load MySQL driver.");
            e.printStackTrace();
            return;
        }

        try
        {
            this.con = DriverManager.getConnection(url, "database_username", "database_password");
            System.out.println("Created a database connection.");
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
    }

    public List<User> getUsers()
    {
        Statement select = null;
        ResultSet result = null;
        List<User> users = new ArrayList<User>();

        try
        {
            select = this.con.createStatement();
            result = select.executeQuery("SELECT username, password FROM users");

            while(result.next())
            {
                // process results one row at a time
                // by mapping the data from the database to
                // the a User object
                String username = result.getString(1);
                String password = result.getString(2);
                users.add(new User(username, password));
            }
        }
        catch (SQLException e)
    }
}

```

```

    {
        e.printStackTrace();
    }
    finally
    {
        closeQueryConnection(select);
    }

    return users;
}

private void closeQueryConnection(Statement select)
{
    try
    {
        if (select != null)
        {
            select.close();
        }
    }
    catch (SQLException e)
    {
        e.printStackTrace();
    }
}

public void closeDatabaseConnection()
{
    try
    {
        if (this.con != null)
        {
            this.con.close();
            System.out.println("Database connection is now safely closed.");
        }
    } catch (SQLException e)
    {
        e.printStackTrace();
    }
}
}

```

Okay, yes I know, I know, it's a lot of code and it's probably quite confusing and maybe even frightening. But that's why I'm here, so let's take the mystery out of this mess shall we?

The first thing you may notice is the (almost ridiculous) use of `try/catch` blocks of code. This is because dealing with databases is sometimes sensitive business. Sometimes things can go wrong with your database operations, and you need to handle these exceptions appropriately. Consider what would happen if you were updating a bunch of values in your database because one of your `Users` changed their address, but halfway through the update, the database connection was dropped. Now your database is left in a state where only half the information was updated for your `User`'s address. Uh oh!

So, when Java was designing their code, they made the decision to force developers to handle these exceptions (they're actually called checked exceptions, as opposed to unchecked exceptions).

In any case, this example code above is littered with `try/catch` blocks because of these checked exceptions. It's also worth mentioning that as your skills develop, you will likely be switching to a framework that will handle all of these complexities so you don't have to. Frameworks like Spring (with their Spring JDBC code) will make dealing with databases much easier!

Okay, so what else is going on here? Well, essentially what's happening is that when we instantiate our `UserDao` object, we create a connection to our database. This connection is what allows us to talk to our database, as we want to tell it to do things like "fetch me some data" or "store this data for me". This is made possible with the `this.con` variable.

The next thing that's happening is inside the `public List<User> getUsers()` method. This method is what will use the `this.con` to query our database, and when I say "query our database", I just mean that it will ask the database to show what data it's storing (we want to **retrieve the data**). So, we retrieve the data (via our `select` statement) and then we iterate through all the results and store the information from the database into some new `User` objects.

That's really all the "magic" that's happening inside of all this code.

1. Open a connection to the database
2. Query the database
3. Store the results in Java Objects

This is really all that ever happens inside of database code in the majority of cases, we always have to make sure we have a connection, then we perform our operation on the database, and then we do something with the results from the operation.

The only real annoying part I find is that you can't just read data from the database and just immediately have it available as a Java Object. There's always a "mapping" that has to be done for each piece of data that's read, from the database to the Java Object. In any case, it's not too complex to implement this mapping, as you've already seen in the code example above (inside the `while` loop).

Step 4: Test the code

Like any good developer, you shouldn't just take my word for it, you should test this code and make sure it works right? So let's create a unit test!

```
package com.howtoprogramwithjava.test;

import java.util.List;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import com.howtoprogramwithjava.business.User;
import com.howtoprogramwithjava.dao.UserDao;

public class UserDaoTest
{
```

```

UserDao sut;

@Before
public void beforeEachUnitTest ()
{
    sut = new UserDao();
}

@After
public void afterEachUnitTest ()
{
    sut.closeDatabaseConnection();
}

@Test
public void testDatabase ()
{
    List<User> users = sut.getUsers();

    for (User user : users)
    {
        System.out.println(user.toString());
    }
}
}

```

Okay, so this isn't exactly the greatest unit test, because we're not using the **Triple A** approach here. We're only Arranging and Acting, we're not Asserting. I did this on purpose because your situation will vary with the data that's available in your database, so doing an assert at the end would likely fail when you run it. All I want to see here is that this code executes without any exceptions. Here's an example of the output when it does run successfully:

```

Created a database connection.

username: testUser, password: testPassword

Database connection is now safely closed.

```

So, for my example, I had one row in my database with a User who had the username 'testUser' and a password 'testPassword'.

Common Pitfalls when Setting up a Database Connection

Now, if you're not getting a green bar when you run this unit test, you'll need to flip over to your console to see what errors are outputted. Some common errors are:

`java.lang.ClassNotFoundException: com.mysql.jdbc.Driver` – This probably means that the program failed when it was setting the driver details. If the String you have that describes your driver is correct (in our case it's "`com.mysql.jdbc.Driver`") then it's possible that you didn't put the driver's JAR

file on the classpath correctly, have you completed [Step 1](#)? If yes, then try copy pasting the error into Google to see if other people have had the same problem as you.

`java.sql.SQLException: Access denied for user 'database_username'@'localhost' (using password: NO)` – This means that you're not putting in the correct credentials to connect to your database. Remember back in [Step 2](#) when I showed you how to setup your MySQL database? You setup a username and password for your database here, be sure to put those details into your code on this line:

```
this.con = DriverManager.getConnection(url, "database_username", "database_password");
```

You need to replace the `"database_username"` and `"database_password"` with the username and password you setup in [Step 2](#).

`com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Table 'testDatabase.users' doesn't exist` – This probably means that you haven't created a database table yet, or if you did, you didn't name it 'users'. This is something we did in [Step 2](#).

Still Can't Figure it Out?

Email me at trevor@javavideotutorials.net. I know how frustrating it can be, and maybe the problem is something simple. Let me know what the error is that you're getting and I'll do my best to help you out.

Chapter 8

Tips to Becoming a Better Programmer

Getting Stuck on a Programming Problem

We've all been there before. We're compiling or running some code and it's just not working. You could be seeing things like Java exceptions in your logs or maybe your IDE is telling you there's something wrong with your code and it's being very cryptic.

How on earth are we supposed to go on from this point? If you're fairly new to programming, you're likely going to be dead in the water until you get some help.

So where can you go to get that help? That's what I'm going to talk about today. I've had tons of questions come in from beginner Java programmers who get stuck in a certain spot and have no idea where to turn for help. Here are the places I would turn to for help:

How to Program with Java.com

If you are following along with this site when learning the Java language, I would love it if you would ask your questions in the comments section on any of these posts. Asking your questions here helps in the following ways:

1. I just sincerely enjoy helping people out.
2. All the other readers of this site will benefit from seeing the answer to your question, as they likely have the SAME question too!
3. Gives me ideas on other topics that I can cover if I notice the same questions coming in.

Stackoverflow.com

Whenever I have been REALLY stuck on a complex programming problem, I have turned to this site with great success. You just ask your question, and there are literally TONS of programmers at your disposal who will try to help you get to the bottom of your problem. Many times I've had developers solve my problems by making me realize that the overall design of my solution needs to be tweaked. To me, this is the best solution to any problem, because if I had just tried to "hack" in a solution to a poor design, I would have WAY more problems in the future. Better to solve the root of the problem instead of the symptoms right?

So, overall, this website is extremely helpful as it can "crowdsource" the proper solution.

Google

This seems like a no-brainer, and for the most part it is. But here's a few tricks that I use whenever I'm trying to solve a problem. I typically solve run-time errors in this fashion:

1. Isolate the exception in your logs/console
2. In your logs/console, find out if you can see the words "Caused By:"
3. Copy/paste the line of words after "Caused By:" into Google and search

This tactic is good for finding related articles on people who have had the same problems as you, and seeing how they went about solving them.

Note that if you don't see any search results, it's very likely that you have some information in the line that you copy/pasted that's specific to YOUR program. See if you can isolate any words that are specific to your program and remove just those words, then do the search again.

What if you don't see "Caused By:"

Well, no worries, the trick I do here is I can the exception until I can see a Class that I recognize. Let's take a look at the following example stack

trace:

```
org.hibernate.HibernateException: No Hibernate Session bound to thread, and configuration does not allo
at org.springframework.orm.hibernate3.SpringSessionContext.currentSession(SpringSessionContext.
at org.hibernate.impl.SessionFactoryImpl.getCurrentSession(SessionFactoryImpl.java:687)
at src.dao.UsersDao.getUser(UsersDao.java:67)
at src.service.UsersService.getUserForUserDetails(UsersService.java:89)
at src.service.UserDetailsService.loadUserByUsername(UserDetailsService.java:35)
at org.springframework.security.authentication.dao.DaoAuthenticationProvider.retrieveUser(DaoAuth
at org.springframework.security.authentication.dao.AbstractUserDetailsAuthenticationProvider.a
at org.springframework.security.authentication.ProviderManager.authenticate(ProviderManager.ja
at org.springframework.security.authentication.ProviderManager.authenticate(ProviderManager.ja
at org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter.attemp
at org.springframework.security.web.authentication.AbstractAuthenticationProcessingFilter.doFil
at org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainPro
at org.springframework.security.web.authentication.logout.LogoutFilter.doFilter(LogoutFilter.ja
at org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainPro
at org.springframework.security.web.context.SecurityContextPersistenceFilter.doFilter(SecurityC
at org.springframework.security.web.FilterChainProxy$VirtualFilterChain.doFilter(FilterChainPro
at org.springframework.security.web.FilterChainProxy.doFilter(FilterChainProxy.java:173)
at org.springframework.web.filter.DelegatingFilterProxy.invokeDelegate(DelegatingFilterProxy.ja
at org.springframework.web.filter.DelegatingFilterProxy.doFilter(DelegatingFilterProxy.java:259
at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.ja
at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:206)
at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:233)
at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:191)
at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:127)
at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:102)
at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:109)
at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:298)
at org.apache.coyote.http11.Http11Processor.process(Http11Processor.java:857)
at org.apache.coyote.http11.Http11Protocol$Http11ConnectionHandler.process(Http11Protocol.java
at org.apache.tomcat.util.net.JIoEndpoint$Worker.run(JIoEndpoint.java:489)
at java.lang.Thread.run(Thread.java:662)
```

This is a problem I recently had with one of the websites I created, and here's how I went about solving it.

1. If you look at the stacktrace error as a whole, it seems very intimidating, so I like to simplify it by just looking at the blue underlined text.
2. I scan them one by one from top to bottom until I find a Class name that I recognize
 - a. **Note:** The first blue underlined text says org.hibernate.HibernateException so this means that the problem has to do with the Hibernate framework... good to know
3. SpringSessionContext.java – Means nothing to me, so I keep looking
4. SessionFactoryImpl.java – Nope
5. UsersDao.java – Aha! I recognize that class, as I created it to save the information for the Users that belong to my program.

6. Now I look at the actual error: No Hibernate Session bound to thread, and configuration does not allow creation of non-transactional one here
7. So I take that error and copy/paste it into Google armed with the knowledge that it has to do with the Hibernate framework and that it's happening in my UsersDao class, which I use to save user information
8. The first result is a question from stackoverflow [click here](#) if you want to see it. And what do you know, I scroll down in the stackoverflow question until I see an answer that was given by a user and it has a green checkmark to the left of it. This denotes that the user's problem was solved by this answer. The person answering the question says that you likely need to include the @Transactional annotation on your class.
9. I put the @Transactional annotation on my UsersDao class to test and see if this fixes my problem... and what do you know, my problem is SOLVED!!!

Trial and Error

I generally go through a few iterations of the Google approach by using a trial and error approach. If the first proposed solution I find on Google doesn't work, I try a few more. If I've been stuck on the same problem for hours, I do something interesting:

Step Away from the Problem!

I can't stress this technique enough. I have wasted so many hours of my life by hammering away at a problem with no success, only to take a break or go to bed, and then come back to the problem and solve it in 5 minutes. It's just the way we're wired as humans, as I've done the same thing with learning to play a song on the guitar or piano, you can practice your butt off and never really play the song very well... but after you take a break and come back, you've made surprising progress. Coding is no different, so **take a break!**

And finally, if I haven't been able to solve the problem over the course of many days, I'll post the question on stackoverflow and always get a solution from someone there... that's just my approach, your mileage may vary, but your secret weapon is that you have ME to help you out should you be frustrated and pulling your hair out!

I hope that helps to shed some light on some troubleshooting techniques, now if you're really serious about learning how to program with Java, I just launched a new website that aims at teaching you Java (as well as other technologies needed to be a real website developer). This tutorial series is very detailed and uses videos as its medium. Unfortunately a subscription to this website is not free, as I had to spend a fair bit of extra money on the hosting of both the website and the videos, and I purchased a neat tablet that allows me to write on the screen using a pen/tablet combination... this gives a real "classroom" feel to the tutorials 😊

If you're interested in learning more (and learning with videos) check out javatutorialvideos.net and sign up for a free trial. Or, if you desperately want in, and you don't want to pay, I'm running a fun

contest to win a subscription to the site! Visit javavideotutorials.net/contest.html for more information.

Unit Testing

I'm very excited to start sharing with all of you, my five best tips to being a better programmer. These tips are ones that I used on my journey to becoming a senior level programmer as quickly as I did. For those of you who are interested, here in Canada, the absolute **minimum** amount of time it takes a programmer to progress to a senior level is five years. I was able to attain my title in five years through hard work, and **smart** work. I know you're all hard workers, but it's time to arm you with some killer knowledge to help give you a strong start!

My first tip revolves around ensuring you have quality code. When creating applications, it's of paramount importance that you keep the number of bugs as close to zero as possible. Now, I say as close to zero as you can, because I realize that once your application grows in size, it's very difficult to have absolutely no bugs. But with this tip, you will certainly be on your way to producing top notch code.

What is Unit Testing?

Unit testing is a way to automate an entire suite of test cases that can be run over and over again in quick succession to ensure that your code is producing expected results. It's a way to break your application up into its most important parts and test them all individually to ensure that every part is working as it should be!

Does that sound simple enough? If not, then no worries, for I have some examples for you. 😊 Let's go back to our example of creating an application that will allow users to login if they enter the proper credentials. The first thing I would do when considering the fact that I wish to unit test a piece of code, is to think about all of the test cases for a particular part of the application.

Here are our requirements:

- 1) A User will be allowed to login to the system when they provide a valid username and password combination.
- 2) If a username is entered that does not exist in the system, and error message must be shown that will alert the user that the username entered could not be found.
- 3) If a username is entered that is valid, but the corresponding password does not match what's on file, then an error message should be displayed explaining that the password is incorrect.

The Application Code

Okay, so now that we have our requirements, let's see some potential code for logging into a system:

```
package com.howtoprogramwithjava.business;

import java.util.ArrayList;
import java.util.List;

public class Login
{
    public Boolean isValidUsername(String username)
    {
        List<User> users = getUsersStoredInSystem();

        // this is called a 'for each' loop, it's
        // similar to a regular for loop, except that
        // it's easier to write, since you don't need
        // to specify an index to start at and end at.
        // it will just loop through all of the items
        // in the 'users' list.
        for (User aUser : users)
        {
            if (username.equals(aUser.getUsername()))
            {
                return true;
            }
        }
        return false;
    }

    public Boolean isValidUsernameAndPassword (String username, String password)
    {
        List<User> users = getUsersStoredInSystem();

        for (User aUser : users)
        {
            if (username.equals(aUser.getUsername()) &&
password.equals(aUser.getPassword()))
            {
                return true;
            }
        }

        return false;
    }

    public List<User> getUsersStoredInSystem ()
    {
        List<User> userList = new ArrayList<User>();

        // create a list of 10 users, each with user names
        // that increment (ie. user1, user2, user3, user4),
        // all these users will have the password set to
        // 'howtoprogramwithjava'
        for (int i=0; i<10; i++)
        {
            userList.add(new User("user"+i, "howtoprogramwithjava"));
        }
    }
}
```

```
    }

    return userList;
}

}
```

Okay, so here we have a Class called `Login` that defines three methods:

- `isValidUsername`
- `isValidUsernameAndPassword`
- `getUsersStoredInSystem`

These three methods will be used to satisfy our requirements. Remember, we must check to see IF a username exists, as well as check to see if a combination of username and password is valid. So, given the code above, how certain are you that this code will work?

Well, the common approach is to test this code by manually typing in some usernames and passwords until something doesn't work right? Well, what if you have 6 different tests that you perform, and on the 4th test, something doesn't work. You would probably find out why it didn't work and then test again right? Well, are you going to re-test every one of the scenarios you already tested before you hit the test case that didn't work? You might, but what happens if the number of total test cases isn't 6, its 600!

Imagine attempting to take almost 600 test cases manually and then finding a bug on test #385. You could easily fix the bug, but now you'll have to go back and test 384 cases before you're confident you didn't break anything else!

In the real world, this kind of testing just **doesn't happen** and this is why some applications you've used had bugs in them. It can happen people, and it's not always pretty! Now, wouldn't it be nice if you could just push a button, and all 600 tests could be performed automatically? Of course it would be great!

This isn't a dream, its reality. It's called unit testing. So let's see how you could go about pulling it off!

JUnit

JUnit is a framework that allows you to create automated test cases that can be run at the push of a button. Now isn't that fancy! Great, you're sold right? Of course you are, so let's see what some unit test code looks like. The first thing we need to figure out, is what our test case should be. Let's take a look back at our requirements:

1) A User will be allowed to login to the system when they provide a valid username and password combination.

Okay, so if a User should be allowed into the system with a valid username/password combination, then that's a great test case to start with. What is a possible combination of username/password that should allow us into our system? Well, since we have a list of 10 valid users (as per our application's code) we should use one of those... I'll randomly choose to test:

username: user3
password: howtoprogramwithjava

Let's see what the code looks like to test this scenario:

```
package com.howtoprogramwithjava.test;

import static org.junit.Assert.assertTrue;
import org.junit.Test;
import com.howtoprogramwithjava.business.Login;

public class LoginTest
{
    // the variable name 'sut' stands for
    // system under test. This is just a
    // coding convention and it refers to
    // the Object we will be testing.
    Login sut = new Login();

    @Test
    public void testValidUsernameAndPasswordCombination ()
    {
        boolean result = sut.isValidUsernameAndPassword("user3", "howtoprogramwithjava");
        assertTrue(result);
    }
}
```

Okay, so this doesn't look too different from what we're used to. Some interesting things to note are the use of the code `@Test` and `assertTrue()`. So let's talk about this stuff!

`@Test`

This is called an annotation, and it's used to 'mark' the method with some extra functionality. When we add this `@Test` annotation to our test method, it will tell JUnit to run our method (kind of like a `public static void main()` method). The big difference here, is that your test class (`LoginTest`) can have **many** test methods with the `@Test` annotation. This means that we can run several methods one after the other in rapid succession! Nice!
`assert True()`

One other important aspect to unit tests is that we need to tell Java what it is that we **expect** to happen once the test is done. So in the case of our test method above, we're testing the case that a valid username and password is given. We would expect that when we invoke the `isValidUsernameAndPassword()` method with a valid username/password combination, we would get a result of `true` right? So since we are **expecting** this method to return `true` we just insert the code `assertTrue(result)` to say that we want to check and make sure that the result is actually `true`. So, this would imply that if the `result` was `false`, we would get some kind of error right? Right! That's exactly what would happen... and JUnit likes to represent this whole 'correct test results' vs. 'incorrect test results' with green vs. red colors!

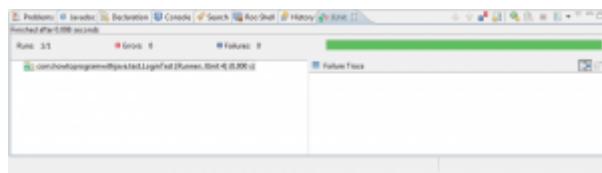
So, without further delay, let's actually RUN our test!

How to Run the Test

This process is very similar to the process of running your programs. All you need to do is:

1. Right click on either the test method or the test Class name
 2. Choose Run As
 3. JUnit Test

If all goes well, you will see a green bar appear that looks something like this:



So, like I said, it will be a green bar if all your tests were successful, and it will turn red if any weren't successful. Also, if there were any that didn't work, it will output the reason why it didn't work. So, let's expand on our test Class and add more test cases based on our requirements:

- 2) If a username is entered that does not exist in the system, an error message must be shown that will alert the user that the username entered could not be found.

So, the test case we should create here is to pass in an invalid username into our `isValidUsername()` method:

```

package com.howtoprogramwithjava.test;

import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import org.junit.Test;
import com.howtoprogramwithjava.business.Login;

public class LoginTest
{
    Login sut = new Login();

    @Test
    public void testValidUsernameAndPasswordCombination ()
    {
        Boolean result = sut.isValidUsernameAndPassword("user3", "howtoprogramwithjava");
        assertTrue(result);
    }

    @Test
    public void testValidUsername ()
    {
        Boolean result = sut.isValidUsername("user231");
        assertFalse(result);
    }
}

```

So, as you can see, we have just added a second unit test to our `LoginTest` Class. This one invokes the `isValidUsername()` method and passes in an invalid username (in this case I randomly chose 'user2834'). This time we use the `assertFalse` piece of code to say that we are expecting this method to return a value of `false`. We are expecting this because we are passing in a username that does NOT exist!

When the test cases are run, you should still see a green bar! So far so good?

Negative Tests

One other important aspect to unit testing is to always test both the positive and negative scenarios. So since we currently have two test cases:

- 1) Test `isValidUsernameAndPassword()` method with a **valid** username/password
- 2) Test `isValidUsername()` method with an **invalid** username

We should also automatically test the opposite of these two outcomes, so we should have tests for:

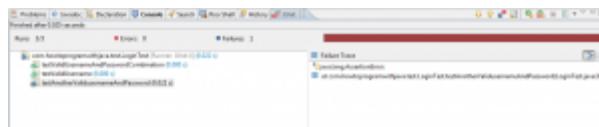
- 3) Test `isValidUsernameAndPassword()` method with an **invalid** username/password
- 4) Test `isValidUsername()` method with a **valid** username

Make sense? I won't include these tests in here, as they'll just take up extra space, and plus it may be more fun for YOU to write those tests yourself 😊

Now, you might be thinking that this code is pretty solid right? We've written a few tests and everything is green and good. What if we introduce this test?:

```
@Test  
public void testAnotherValidUsernameAndPassword()  
{  
    boolean result = sut.isValidUsernameAndPassword("User2", "how to program with  
java");  
    assertTrue(result);  
}
```

This test (as is stated in the name of the test method), should be testing another valid username and password combination. But, this time the test FAILS! What the heck happened? Here's a screenshot of the failure.



Well what the heck happened here?

Upon closer inspection, you may notice that I accidentally made the username "User2" with a capital "U" instead of a lowercase "u." This is very interesting. How should our system react when you give uppercase vs. lowercase letters in the username? This is a result that may not have been caught with regular testing, or if it was caught, it may have taken a while to reproduce if you had accidentally typed in an uppercase letter. This is the beauty of unit testing, it's easy to re-create a failing test!

So what do we do? If this was the real world, you'd likely have to re-read the specifications to see if they tell you what to do in this case, and if the requirements aren't detailed enough, then you'd likely talk to the business analyst who helped write that part of the requirements document. In this case, I'm the one who wrote the requirements, and I purposely made them a little vague around case sensitivity.

So, in this case I'll just say that the **password needs to be case sensitive** and the **username should not be case sensitive**.

Let's go back and change our application code with these new requirements:

```
public class Login
{
    public Boolean isValidUsername (String username)
    {
        //... content removed for brevity

        // here we use equalsIgnoreCase() instead of just equals()
        if (username.equalsIgnoreCase (aUser.getUsername ()))
        {
            return true;
        }
        //...
    }

    public Boolean isValidUsernameAndPassword (String username, String password)
    {
        //... content removed for brevity

        // here we use equalsIgnoreCase() instead of just equals() on the username
        // but we leave the equals() comparison for the password!
        if (username.equalsIgnoreCase (aUser.getUsername ()) &&
password.equals (aUser.getPassword ()))
        {
            return true;
        }
        //...
    }

    // ...
}
```

So, now that we've identified a real bug and we've changed our application's code to fix the bug. Let's go back and re-run our tests!

If all goes well, you'll see a green bar and breathe a sigh of relief.

Now, I'm sure there are a bunch of other test cases that you could think up. I know it's always useful to test with `null` values to ensure your code won't throw any unwanted exceptions. Why don't you try to write some tests that use `null` for the values of `username` and `password`? I think you'll see some interesting results. 😊

Summary

Automated testing (via JUnit or any other testing framework) is the wave of the future. All applications should be built on a solid foundation of unit tests to ensure that everything is still working the way it's

expected to work as the product evolves. It's the difference between happy customers and angry customers. So, it's absolutely worth the effort to create unit tests to avoid these headaches. 😊

Don't Be Clever

In my years as a programmer, I've come across some really really smart people who could create some very fancy code that (at the time) seemed to be the greatest stuff since sliced bread. But let me tell you, now that they have moved on in their careers and have left the company at which I currently work, I look at this code that **I** now have to support, and I shake my head.

This code is too complicated!

What a nightmare, what first seemed like the level of coding talent I aspired to reach, now became the bane of my existence. So how could this be? How could I look at one piece of code one day and sing its praises, then the next be stating that it's causing me grief? I'm not sure what name to give to this other than the loss of context. When you are creating a particular piece of a software application, you are very familiar with all the bits and pieces of that part of code, because you're in the process of creating it. But if you allow some time to pass, you'll soon forget all of the ins and outs of the code you've already written. This will mean that if there's a bug in your code, you (or some other poor soul) will have to **re-learn** what that code does. So if that code isn't easy to read, interpret and follow, then you may end up pulling your hair out!

Keep It Simple Stupid (KISS)

This is the motto you should keep in mind when writing and designing your applications. Here's a wonderful quote that lends itself well to this motto:

Any fool can write code that a computer can understand. Good programmers write code that humans can understand. – Martin Fowler

Writing code that can be understood by your fellow programmers is key, which is why if you're writing some code that you feel is a little complex when you're writing it, ask yourself

“If I came back and looked at this code in three months, would I be able to easily understand what's going on?”

If the answer to this question isn't a resounding and immediate “YES”, then you should take action to change your answer **to** an immediate and resounding “YES”.

How do I Keep it Simple Stupid?

Comment your Code

One way to make sure your code is easily understood by all, is to put in some comments that explain what any particular block of code is actually doing. This is fairly effective, but doesn't exactly guarantee that your code will be understood by all. Take this code snippet for example:

```
for( i = 0, j = 100; i < j; i++, j--)  
    System.out.println(i);
```

What in the heck is this piece of code doing!? You could probably sit around for 10 minutes trying to figure it out, but, if I just add some comments before the `for` loop, I could save you a lot of grief:

```
// count to 49  
for( i = 0, j = 100; i < j; i++, j--)  
    System.out.println(i);
```

Oh, I see! They are just counting all the way up to 49 and outputting each number in the console. Fair enough. This is effective, but what if someone were to change the code inside that `for` loop and not update the comment to reflect what it is they've changed? This is the downfall of relying on comments, they aren't always correct, and sometimes that can be more damaging to your debugging efforts than if they just hadn't commented the code at all.

But in any case, for the most part this tactic is useful and should be followed by developers. But, I've got one more neat way to ensure that your code is simple and easy to follow... and you may be surprised to hear it.

Unit Testing!

Believe it or not, I find that when I'm writing code and creating unit tests at the same time, this helps to keep the code simple. Why is that you ask? Well, I find that when I'm writing unit tests, the code **has to be simple** in order for it to be unit testable in the first place. The longer and more coupled a piece of code is, the harder it is to unit test. So if you write your code while unit testing it at the same time, a beautiful thing happens. Your code becomes less coupled, each method tends to keep to a reasonable size AND you've got a whole bunch of unit tests that help describe what your code does (through utilization of descriptive unit test methods, and expected results with your `assert` methods).

Descriptive Variable Names

I cannot overstate the importance of being explicit with the names of your variables. Let me tell you the pain and eye gouging that goes on when I see a block of code that looks like this:

```
Date date1 = new Date();
```

```

Date date2 = x.getDateFromY();
Integer temp = 0;
Integer y = 30;

while (date1 < date2)
{
    temp++;
    if (y < temp)
    {
        date2.setDay = x.getDay();
    }
    else
    {
        date1.setDay = y;
    }
}

```

Seriously!? What the hell is going on here! It drives me mad when I see people naming their Date variables date1 or date2, just name your variable after what it truly is! If I was assigned to debug this code, I wouldn't have a clue what the expected behaviour should be... but had the developers used more descriptive variable names, like effectiveDateOfPurchase instead of date2 and currentDate instead of date1 and lastDayOfMonth instead of y, I may have had a better idea of what's going on here.

Note: This particular block of code doesn't do anything, I literally just made it up on the spot to get my point across.

So, the point I'm trying to make here, is that you shouldn't be afraid of using a long variable name. Java won't get mad at you for having a variable name that's 20 characters in length. Plus, modern IDEs make it easy to track variables and rename them if they really do get too out of hand. But I am a firm believer that no seasoned developer would get upset with you if you were to use a descriptive name for your variables, because it really does help people understand what the code really does!

Summary

So, the most important thing you should take away from this discussion, is that you should KISS... Keep it simple stupid. Comment your code when you think people may get confused as to what it's doing. Unit test your code so there's less chance of your methods becoming unwieldy. And please, PLEASE, be descriptive when naming variables/methods. The world will **literally** be a better place if all developers followed these simple tips.

Refactoring Tools

This tip is something I only learned about in the last year or so of my programming career. But since learning it, I find that it's something I just can't live without and am constantly sharing with other developers.

What is Refactoring?

Let's take a look at Wiki's definition of refactoring:

Code refactoring is a “disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior”, undertaken in order to improve some of the nonfunctional attributes of the software. Typically, this is done by applying a series of “refactorings”, each of which is a (usually) tiny change in a computer program’s source code that does not modify its conformance to functional requirements. Advantages include improved code readability and reduced complexity to improve the maintainability of the source code, as well as a more expressive internal architecture or object model to improve extensibility.

Now normally I'm not a fan of these definitions because they're a little too technical and attempt to be as "complete" as possible to please everyone. But in this case, I think this is a great definition!

Refactoring is the process of taking existing code and changing it in some way to make it more readable and perhaps less complex. The key thing to note, is that when the code is changed, it does **NOT** affect the underlying functionality of the code itself. So the changes you're making are literally just to make the code easier to follow by your fellow developers.

Okay, so now that you know what refactoring is all about, how can you go about refactoring the code you already have?

Refactoring Tools

The SpringSource Tool Suite (and Eclipse) provide some great refactoring tools that are easy to use. So, let's take a look at an example of some code that could use some refactoring shall we:

```
import static org.hamcrest.CoreMatchers.is;  
import static org.junit.Assert.assertThat;
```

```

import org.junit.Test;

import com.howtoprogramwithjava.business.Card;
import com.howtoprogramwithjava.business.Dealer;
import com.howtoprogramwithjava.business.Deck;

public class DealTest
{
    @Test
    public void testDealingOfCards ()
    {
        Deck deck = new Deck();
        Dealer.deal(deck.getDeck(), deck.getPlayers());

        for (int i=0; i<deck.getPlayers().size(); i++)
        {
            assertThat(deck.getPlayers().get(i).getCardsInHand().size(), is(5));

            System.out.println(deck.getPlayers().get(i).getName() + " has the following cards:");
            for (Card aCard : deck.getPlayers().get(i).getCardsInHand())
            {
                System.out.println(aCard.toString());
            }
            System.out.println();
        }
        assertThat(deck.getDeck().size(), is(32));
    }
}

```

This code was used to unit test the dealing of cards to four players. We are ensuring that every player has 5 cards after the deal, and that there are only 32 cards left in the deck (after 20 cards were dealt out to four players). We also list out what cards are in each players' hand.

Now, as I stated in the beginning, refactoring code isn't about changing its functionality, but to make the code a little bit more readable. So to do this, the first rectoring I want to do is to extract local variables.

How to extract Local Variables

This process will involve us taking a commonly occurring or repeating piece of code and assign it to a variable so that we don't have to keep writing out the code to reference it every time.

Okay, so maybe that doesn't make a whole lot of sense to you just yet, but how about we identify in the above code where we have some repeating references. Look how many times we write out the code `deck.getPlayers()`... 5 times! Wouldn't it be easier to just assign that to a local variable named `players`? For sure! So how do we do that?

1. select the code `deck.getPlayers()`
2. right click on it -> choose Refactor
3. choose Extract Local Variable

Note: All references to the exact ways of using the refactoring tools is only applicable for Eclipse and STS, any other IDE will have a different set of menus to run the same refactoring techniques

Also Note: You can use the shortcut key sequence Alt-Shift-L to extract a local variable

Okay, so now you're presented with a screen that asks you what you'd like to call your new local variable, I just **leave it as the default** players name and **hit enter**.

Now let's take a look at our new code:

```
@Test
public void testDealingOfCards ()
{
    Deck deck = new Deck();

    // here's our new local variable
    List<Player> players = deck.getPlayers();
    Dealer.deal(deck.getDeck(), players);

    for (int i=0; i<players.size(); i++)
    {
        assertThat(players.get(i).getCardsInHand().size(), is(5));

        System.out.println(players.get(i).getName() + " has the following cards:");
        for (Card aCard : players.get(i).getCardsInHand())
        {
            System.out.println(aCard.toString());
        }
        System.out.println();
    }
    assertThat(deck.getDeck().size(), is(32));
}
```

Look at that! Now all of the places in the code where it used to say `deck.getPlayers()` now points to the `players` local variable, and it's automatically assigned `deck.getPlayers()` **to this players local variable**.

That was pretty painless right? Now, what else can we do to clean things up? I see two references to `players.get(i).getCardsInHand()`, so let's apply the same technique again to create a local variable called `cardsInHand`.

How to Extract a Method

Now that you're familiar with extracting a local variable, how about we do the same kind of thing, only with an entire method! This particular refactoring tool will take an entire block of code and turn it into a method. This is very useful for cutting down the number of lines in any given method, and actually helps to self-document the method (thus making it more readable).

Let's see if there are any blocks of code in our example that could be extracted into a method...

I see that there's a chunk of code that's just dedicated to outputting details to the console... and this could be thought of as just "noise" in the testing method, so how about we extract it into another method 😊

This is a very similar process to extracting a local variable, all you must do is:

1. select the block of code you wish to extract
2. right click on it -> choose refactor
3. choose Extract Method

Note: This can also be done by selecting the code and hitting Alt-Shift-M

Now you'll be presented with an "extract method" screen that asks you to name the method you're about to create. Well, since this is just a block of code that outputs the cards to the console, let's **name the method** `outputCardDetailsToConsole()`. So type that in and **hit Enter**.

Let's take a look at what we've done now:

```
@Test
public void testDealingOfCards ()
{
    Deck deck = new Deck();
    List<Player> players = deck.getPlayers();
    Dealer.deal(deck.getDeck(), players);

    for (int i=0; i<players.size(); i++)
    {
        List<Card> cardsInHand = players.get(i).getCardsInHand();
        assertThat(cardsInHand.size(), is(5));

        outputCardDetailsToConsole(players, i, cardsInHand);
    }
    assertThat(deck.getDeck().size(), is(32));
}

private void outputCardDetailsToConsole(List<Player> players, int i, List<Card> cardsInHand)
{
    System.out.println(players.get(i).getName() + " has the following cards:");
    for (Card aCard : cardsInHand)
    {
        System.out.println(aCard.toString());
    }
    System.out.println();
}
```

Would you look at that, now in our test method, all those noisy lines of console output code have been reduced to one line of code that says `outputCardDetailsToConsole(players, i, cardsInHand)`. The IDE has automatically figured out what parameters need to be passed into the new method that was created and passed in those variables for you. Absolutely painless right? Now the code is much more readable!

Rename Variables

Now there's one shortfall of the "extract method" refactoring tool, and it's that it will often choose non-descriptive variables names for the parameters in the method's signature. So how about we change one of the variable names using our refactoring tools!

I don't like the fact that it has used the variable name `i` for the player number, so in order to change it, we just:

1. select the variable we want to rename
2. right click -> choose refactor
3. choose rename

You'll see that the variable gets outlined in blue and there's a little dialogue that appears that states "enter new name, press **Enter** to refactor". So that's exactly what we'll do! Type in "playerNumber" and hit **Enter**.

Voila, we've now renamed the variable in the method's signature as well as every occurrence of that variable in the method itself. PAINLESS people!

Note: You can rename a variable using the Alt-Shift-R shortcut sequence.

For all of those shortcut key maniacs out there (like myself), you'll notice that all of your refactoring tools are available via the Alt-Shift-(something) sequence. So just always remember that **Alt-Shift** is your refactoring friend 😊

How to Change a Method's Signature

Now let's say that we've decided that we want to add to our console output. We've decided that we also want to output how many cards are left in the deck... but we don't have the necessary information available to us in our new `outputCardDetailsToConsole()` method. So we'll need to change the method signature and add a new parameter. Now we could do this manually in this case fairly easily by just adding in the parameter ourselves, but, the whole point here is that the IDE will automatically update ALL the references to that method. Imagine a scenario where we have 100 different methods referencing the one we need to change, that would be a big pain in the butt to change manually, but with the "Change Method Signature" refactoring tool, it's a piece of cake.

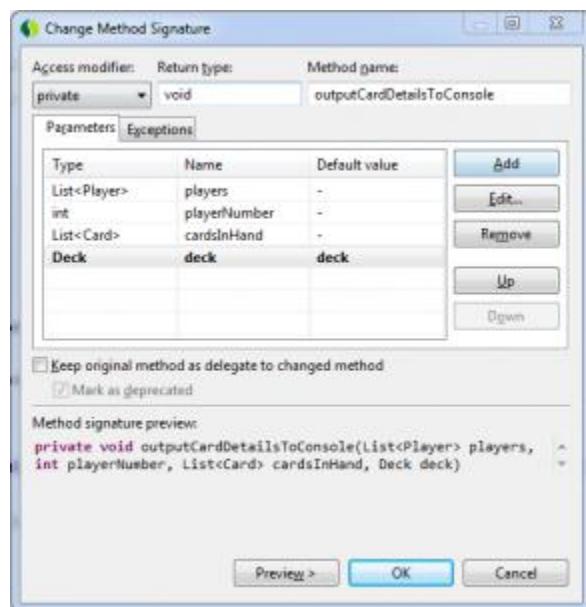
1. Select the method's name that you wish to change
2. Right click -> Choose refactor
3. Choose Change Method Signature

Now, we want to pass in the `Deck` object, so in the “Change Method Signature” dialogue that appears:

1. click the “Add” button
2. Type `Deck` in the “Type” column
3. Give it a good name, like “`deck`”
4. For the default value, type in “`deck`” as well

Note: The “Name” column represents the name of the variable as it appears in the method you’re changing. The “default value” column represents what variable name it should put in for where this method is being called, in this case it’s being called from our test method, and we know that the test method has a `deck` variable, so it’s safe for us to use this “default value”, if you’re unsure what value to use, you could put in `null` and then go back and change all the references where needed.

Here’s a screenshot of what the “Change Method Signature” should look like after you’ve entered your details:



Finally, here’s what your code will look like after we make use of this added `deck` variable:

```
@Test
public void testDealingOfCards () {
    Deck deck = new Deck();
    List<Player> players = deck.getPlayers();
    Dealer.deal(deck.getDeck(), players);

    for (int i=0; i<players.size(); i++) {
        List<Card> cardsInHand = players.get(i).getCardsInHand();
        assertThat(cardsInHand.size(), is(5));

        outputCardDetailsToConsole(players, i, cardsInHand, deck);
    }
}
```

```

        }
        assertThat(deck.getDeck().size(), is(32));
    }

private void outputCardDetailsToConsole(List<Player> players, int playerNumber,
    List<Card> cardsInHand, Deck deck)
{
    System.out.println(players.get(playerNumber).getName() + " has the following cards:");
    for (Card aCard : cardsInHand)
    {
        System.out.println(aCard.toString());
    }
    System.out.println("The deck has " + deck.getDeck().size() + " cards in it.");
    System.out.println();
}

```

Summary

I use the refactoring tools on a daily basis at my job and I find that it saves me tons of time. Every time I'm pair programming with someone and they happen to see me use one of these tricks, they always stop me and ask me what it was I just did. Then they end up thanking me for saving them so much time... I should seriously start charging for all these tips and tricks 😊

So, use these tricks and remember that the **Alt-Shift** keys are your refactoring friend! I look forward to seeing some more readable code out there everyone 😊

Challenge Yourself

When I first started working at my current job, I was just barely out of the gates as a programmer. I had about a year's worth of experience under my belt, and was being thrown into a new workplace as an intermediate level developer. I managed to do well enough in the interview to fool the company into thinking I was an intermediate developer, when I was definitely a junior level.

So, in the hopes to rectify this situation, I took note of what frameworks the company was using for their web application product and set out to learn them.

Continuous Education

It wasn't mandatory for me to learn the framework the way I did, but I knew that the best way to learn something is to throw yourself into it head first.

So my tip for you today is all about learning something new by choosing to learn it in a way that you'll have to dedicate yourself to learning it. My example of this is something called the Spring framework (it's created by the same people that made the SpringSource Tool Suite IDE that you may be using). The Spring framework is used to help you when building a web application, its uses are a plenty, but that's not what I want to talk about in this article.

The point here is that there was a particular piece of technology that I needed to learn, and at the same time there was a project that I wanted to startup with my business partner. This project I wanted to start was a web application that would help people in the ecological land classification industry. So I figured, what better way to kill two birds with one stone than to use this Spring framework for my new project? This would force me to learn the technology while I created this web application on the side, and it would benefit my professional career as well!

The Best Decision I Made

As a result of this decision, I embarked on the journey of learning the Spring framework, and let me tell you, I learned a ridiculous amount! In order to build this web application, I would need to learn and understand the vast majority of this framework. The most important part of this entire process, was that I was being held accountable to produce results (by my business partner), so I didn't have the opportunity to just give up on learning.

Now you may not be in the same situation that I was, you could be trying to learn the Java programming language, a different framework within Java, or perhaps an entirely new language altogether... But in

any case, I think it's important to have someone that will hold you accountable for your learning. Now, if you don't have any ideas for any projects that you would like to actually release and start selling, try and think up a project that you will **want** to complete. For some programmers it's the challenge of creating a game, or a fun website (like a blog). What matters is that it should be something that you will enjoy working on.

Adapt to Changing Times

Another good example of this concept of challenging yourself to learn, was when I decided I wanted to learn a new programming language. I looked at the languages out there that had some momentum behind them, and decided I wanted to learn Objective C. This language is what is used to create iPhone apps on the iPhone/iTouch/iPad. There seemed to be a lot of hype around creating mobile apps, so I figured I'd think up an app I could actually use myself.

I realized that there was a need in the iPhone market for an app that would automatically text message my girlfriend and let her know that I was almost home (or that I had left work). I was tired of sending text messages while trying to drive, so wouldn't it be neat if there was an app that just automatically monitored your location and sent a text message to a person of your choosing between x/y o'clock?

So once again, I was in a win/win situation here. I would be creating an app that I wanted to use, I could then sell the app, and the worst case scenario was that I would learn how to program in Objective C (which is a great skill to have). I had my girlfriend hold me accountable for the creation of the app by monitoring my progress (she was very supportive), and within a few months I had completed the app! It's quite fun to be able to show my friends that I had an app in iTunes!

Serendipity

It was interesting to me how much I actually learned about Java when I was learning about the Objective C language. This was mostly because both languages share roots in Object Oriented programming. Nevertheless I hadn't intended on expanding my knowledge in Java when I set out to learn Objective C, but that's exactly what ended up happening. This fact presented itself in the form of understanding the bigger picture and the foundations of any programming language. When you learn multiple programming languages (or even spoken languages), you can't help but identify similarities between them and in turn understand the foundations on which they're built.

I've turned this concept into overdrive when I started this "How to Program with Java" blog. I'll be the first to say that you learn an incredible amount of things when you try to teach a subject. This is mostly because I want to try and find the best way to explain a particular topic, and in order to do so, I need to understand all the inner workings of my subject. Thankfully, I find this adventure to be very fun and rewarding! Helping people is a business everyone should love 😊

Summary

So overall, I've found that I've been able to "level-up" my skills over and over again by following this one simple tip.Challenge yourself! But do so in a manner that makes it hard for you to give up. Find someone to hold you accountable. I hope that **YOU** will be holding **ME** accountable for the content that I provide to you.

When you are continually learning, you will find yourself surpassing most of your peers at (what I considered) alarming speed!

Test Driven Development



This concept was first introduced to me around October of 2010 by a group of coaches that were hired to train employees in companies how to follow a software development approach called Agile. Let me be clear when I say:

When I used the Test Driven Development (TDD) approach, I had never written code that more stable (bug free) and easy to maintain. Period. End of discussion.

As you can see, I am very passionate about this topic, only because I saw the results first hand. The particular piece of the application that I build using the TDD approach was a large section that deal with a lot of complex scenarios, so it was an ideal candidate to use the Test Driven Development strategy.

So what is Test Driven Development?

The premise is that you **write the unit tests for your code first**, before any actual code to solve the problem you're currently working on.

Let's pull a definition from wiki:

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and INSPIRES CONFIDENCE.

When I was first introduced to the concept of TDD, I didn't know what to make of it, it just seemed crazy to me that you would write a test case first without any real code to test. Why on earth would you write a unit test for a block of code that didn't exist? You know the test is going to fail, so what's the point?!

Well, first off, it's all about the **repetition** of a **short development cycle**. When you write a failing test, you are outlining a requirement that needs to be implemented. So this will force you into actually completing a step that most developers skip: the documentation!

This may not seem like a big win to some, but it certainly paid off when the code went live on the internet. We were getting complaints from certain customers about this piece of functionality, and

since I was the one who worked on it, the complaint landed on my desk. I reviewed the customer's complaint and said "Wait a second, this customer is complaining about a bug that's NOT a bug, that's just how we designed the code to work". I quickly pulled up the unit tests that were written to review whether or not I was correct, and sure enough, that's how we **designed** it to work! So I:

1. Confirmed with a BA that the requirements were incorrect
2. Had the requirements updated
3. Updated the test case (to reflect the correct outcome)
4. Made the appropriate code change
5. Re-ran all the unit tests to confirm nothing else broke

After this was done, a wonderful thing happened. Since I had such a large volume of unit tests around this particular piece of the application (**due to my TDD approach**), I was **extremely** confident in my code change. So I checked in the code change, assigned it to the QA team, and sure enough... no bugs!

I find that having the unit tests created first (as part of TDD) ensures that all your code will be documented against the requirements, so things (like a mistake in requirements) are easily addressed. And then if a change needs to be made to the code, it can be made with a high level of confidence that you didn't break anything else.

Okay so this sounds great, but how can I start?

Well, this is a three step process:

1. Write a failing test
2. Write the minimum amount of code to make the test pass
3. Refactor your code

You just keep repeating these steps one by one until you've implemented all the necessary requirements. So, how about we outline some requirements? This will allow us to put our new TDD skills to the test!

REQUIREMENTS

Our customer needs an application built that will count the number of words in a paragraph. There should be a count of the total number of words in the paragraph as well as a count of the number of times an individual word is used in the paragraph (they want to see what words are most commonly used).

IMPLEMENTATION

Ok, so let's start with our Test Driven Development. The first thing we need to do is write a failing test. Our first test could probably just count the number of words in a sentence (by counting the number of spaces in a sentence).

WordCountTest.java

```
import static org.junit.Assert.*;
import org.junit.Test;

public class WordCountTest
{
    WordCount wordCount = new WordCount();

    @Test
    public void countTheNumberOfWordsInASentence ()
    {
        String sentenceWithFourWords = "How are you today?";
        Integer countFromString = wordCount.getWordCountFromString(sentenceWithFourWords);
        assertTrue(countFromString == 4);
    }
}
```

WordCount.java

```
public class WordCount
{
    public Integer getWordCountFromString (String sentence)
    {
        return 0;
    }
}
```

Okay, so when we create the `WordCountTest.java` we create a test method with an appropriately descriptive name (ie. `countTheNumberOfWordsInASentence`), then the contents of this method will invoke our system under test. The system under test in this case is a new Class we had to create called `WordCount.java`. I then created a very simple method inside of the `WordCount.java` file that just returns the value of zero for the time being. We then make sure that the returned result is what we would expect it to be, given our inputs. So, since our input was a four word sentence, we should expect the System Under Test to return the value of 4! But, since we haven't implemented the code inside of our system under test, we get a failing test. Missing accomplished. (If you don't remember how to run a unit test, just [click here](#))

Note: The key to setting up a good unit test is to follow the **Triple A** rule:

- **Arrange** – This is where you will be setting up all of the variables in your test case
- **Act** – This is where you will be performing the actual call to the System Under Test (SUT)
- **Assert** – This is where you will test to make sure everything is as expected

Now that we've implemented the first step of our three step process for Test Driven Development (write a failing test), we move on to step 2... Write the minimum amount of code to make the test pass. So let's try and implement some real code to make our test pass! We'll just change the method in our WordCount class

```
public class WordCount
{
    public Integer getWordCountFromString(String sentence)
    {
        String[] stringArray = sentence.split(" ");
        return stringArray.length;
    }
}
```

With this code change above, we now have a passing test. All we needed to do was use the `split()` method, which will split up a single `String` into an `Array` of `Strings` based on a specified character. So all I had to do was supply a blank space (" ") as the character to split up the `String`. This then created an array of `Strings` and stored it into the `stringArray` variable. We then return the length of our new `stringArray` variable, and voila, we have the length of 4!

So, now we've completed step 2, now all that's left is to refactor our code. But in this case, the code is already pretty simple and easy to read. We could maybe add a check to make sure that the `sentence` variable is not null, but perhaps we could just add that as a test instead. So for now we'll skip step 3 and instead repeat the cycle.

Let's go back to step 1 and add another failing test. Keeping in mind our Triple A rule, we should follow the structure of Arrange, Act and Assert, like so:

```
import static org.junit.Assert.*;
import org.junit.Test;

public class WordCountTest
{
    WordCount wordCount = new WordCount();

    @Test
    public void countTheNumberOfWordsInASentence ()
    {
        String sentenceWithFourWords = "How are you today?";
        Integer countFromSentence = wordCount.getWordCountFromString(sentenceWithFourWords);
        assertTrue(countFromSentence == 4);
    }

    @Test
    public void testThatNullValueForSentenceReturnsZero ()
    {
        // Arrange
        String sentence = null;
```

```

    // Act
    Integer countFromString = wordCount.getWordCountFromString(sentence);

    // Assert
    assertTrue(countFromString == 0);
}
}

```

This time when we run our two tests, the first one still passes, but the second one fails. This is what we were hoping for, as our goal was to write a failing test. So when we take a look at **why** our test failed, we see that it's a `NullPointerException`:

```

java.lang.NullPointerException
at com.howtoprogramwithjava.test.WordCount.getWordCountFromString(WordCount.java:7)
at com.howtoprogramwithjava.test.WordCountTest.testThatNullValueForSentenceReturnsZero(WordCountTest.java:26)

```

This is happening because when we pass in null as an input to our method, it will try to invoke `sentence.split(" ")`, but sentence is null, so we get a `NullPointerException`. The key thing to remember here, is that when you try to invoke null (dot) anything, you get a `NullPointerException`.

So now we repeat step 2, which is to write the minimum amount of code to get our tests to pass, like so:

```

public class WordCount
{
    public Integer getWordCountFromString(String sentence)
    {
        if (sentence == null || sentence.length() == 0)
            return 0;
        String[] stringArray = sentence.split(" ");
        return stringArray.length;
    }
}

```

With this new code, the tests will pass! Now we can perform step 3, refactoring! By the looks of it we have a validation step at the beginning of our method that will make sure to handle some special situations (like when the `sentence` is `null` or empty). I think that looks a little ugly and we can probably use our refactoring tools to extract a method from this. So let's do that!

```

public class WordCount
{
    public Integer getWordCountFromString(String sentence)
    {
        if (isValidString(sentence))
            return 0;
        String[] stringArray = sentence.split(" ");
        return stringArray.length;
    }

    private boolean isValidString(String sentence)
    {
        return sentence == null || sentence.length() == 0;
    }
}

```

As you can see above, we've extracted the validation stuff out into a method called `isValidString()` which will return `true` or `false`. If it's `true`, then we return a 0 `Integer`, otherwise we allow our processing to continue. And upon running of the tests again, we see that we still have a green bar (all our tests pass). Excellent, let's repeat our TDD steps.

Our next requirement is to count the number of times a particular word appears in a sentence. This sounds like a great time to use some `Collections`, perhaps a `Map` will suite our needs. Let's first write a failing test for this situation:

WordCountTest.java - snippet

```
@Test
public void shouldReturnACountOf3ForWordIn ()
{
    // Arrange
    String sentence = "This sentence is designed to have a few repeating words in it, "+
        "like the words is, as well as in, and the word words. As you can see I have also "+
        "included a fair number of spaces in here as well, so we should make sure we don't "+
        "include the space as part of a word that repeats!";

    // Act
    Integer countFromString = wordCount.countNumberOfTimesAnIndividualWordAppears(sentence,
        "in");

    // Assert
    assertTrue(countFromString == 3);
}
```

WordCount.java - snippet

```
public Integer countNumberOfTimesAnIndividualWordAppears(String sentence, String wordToCount)
{
    return 0;
}
```

Now, as you can see, this test has a nice lengthy sentence to test. Our test just calls a new method that passes in our sentence as well as the word we wish to count. This test will fail, as we expect to see three occurrences of the word "in", and the default implementation we've created just returns 0.

Let's move onto step 2 and make sure our tests pass:

```
public Integer countNumberOfTimesAnIndividualWordAppears(String sentence, String wordToCount)
{
    if (isValidString(sentence))
        return 0;

    sentence = stripAllNonAlphaCharacters(sentence);

    int count = 0;
    int position = 0;

    wordToCount = " " + wordToCount + " ";
    while (sentence.indexOf(wordToCount, position) != -1)
```

```

    {
        position = sentence.indexOf(wordToCount, position) + 1;
        count++;
    }
    return count;
}

private String stripAllNonAlphaCharacters(String sentence)
{
    return sentence.replaceAll("[^a-zA-Z ]", "");
}

```

This new code will now enable our tests to pass. You may look at this code and scratch your head wondering what this `stripAllNonAlphaCharacters` method does. Rightfully so, this method is using the `replaceAll()` method which makes use of regular expressions. What this does is essentially look at all the characters in our sentence, and if the character is not the letters a through z, or capital A through Z (or a space character) then it will replace that character with a blank space. I did this so that I could remove all the commas, periods and any other punctuation in the sentence. This is needed in order to properly match whether or not we have found an instance of our `wordToCount` (which in this case is "in").

I also make use of the `indexOf()` method, which will search a `String` and return the numeric position of the `String` that you're searching for. If you like, try to debug this code and follow how the `position` variable changes each time it iterates through the `while` loop.

And finally what must we do now? I'm sure you've guessed it, we do step number 3, which is refactoring. I'm sure by now you get the idea here. So congratulations, you've read through and (hopefully) understood the concepts behind this thing we call Test Driven Development. It's something that I simply love doing, because it gives me absolute **confidence** in my code and allows it to be **modified with confidence** as well! Very powerful stuff in the real world 😊

How to Create a Web Application

A beginner's guide

The following chapters in this book are some exclusive bonus content that you have received because you bought this book via <http://javapdf.org>

Bonus Content Chapter 1

Things you need to Know

What is a Web Application?

The perfect examples of web applications are the social network web applications like Facebook or Twitter. These social platforms are web applications because they exist on the web. You need to navigate to facebook.com or twitter.com in a web browser in order to use them. You will also need to login with a username and password, and this implies that there is some sort of database backing the entire login process.

So in essence, anything that you need to access via a web address (through a web browser like Internet Explorer, Google Chrome or Mozilla Firefox) that has some sort of database backing the program can be considered a web application.

Why are Web Applications so Great?

The main benefits of using a web application are as follows:

- Global access to the application
- Instant delivery of bug fixes
- Cross platform compatibility

Global Access to Application

If you compare a web application (like Facebook) to a common desktop application (like Microsoft Word) you'll realize that you cannot access Microsoft Word from anywhere in the world. If you were to walk into a Starbucks with just a cellphone, you won't be able to access your MS Word documents from your computer at home and start modifying them. However, with web applications, you'll be able to navigate to the web address with your mobile device and get access to all the information just like you would if you were at home.

Instant Delivery of Bug Fixes

When you're working with web applications, all your customers will have to go to your webpage before they can access your application. What's neat about web applications, is that you can choose to have just ONE version of the software live on the internet at any given moment. So this means that when your customers access your web application, they will ALL be using the same version at the same time. The advantage to this is that if you have bugs in one version, you can fix the bugs and immediately distribute that fix to all your customers at the same time. This is not possible with desktop applications, as you'll either need to ship physical CDs/DVDs to the customers to install upgrades, or they will need to choose to upgrade their version whenever they so desire.

Cross Platform Capability

Since web applications are viewed by using web browsers, any web application will work on any operating system (PC/Mac, iOS/Android). Cross platform capability means you're not limiting your potential client-base, which is always a great thing in business!

What Technologies are used when building Web Applications?

In this book, we will be focusing on the Java programming language. It is used as the backbone when building web applications. Other useful technologies that are used in conjunction with Java, which we will be using, are:

- Spring Framework
- Hibernate
- MySQL
- HTML
- Javascript
- JSP (JavaServer Pages)

Now, all of these technologies could have a book written about them individually, so the scope of this eBook will be to show you how they are actually used in practice and to teach what you need to know to get the job done. Having said that, here's a quick breakdown on each technology so you are not completely lost!

Spring Framework

This is used as a means to make our lives easier when we create web applications. The Spring Framework is not required to build web applications, but it helps make the code neater and automatically enforces the use of design patterns.

Note: If you don't already know, think of a design pattern as a 'best practice'. Over the course of programming, certain best practices have shown themselves as being advantageous to follow; these best practices have been formed into what's now called design patterns.

The main purpose of the Spring Framework is to **reduce coupling** of your objects and make use of the **MVC** (model-view-controller) **design pattern**.

What's Coupling?

When two objects are tightly coupled, it means that one is dependent on the other (or worse, they're both dependent on each-other). An example of this can be seen when one Object is instantiated inside of another Object:

```
public class User
{
    private String username;
    private String password;
```

```

private UserAddress userAddress;

public User(String username, String password)
{
    this.username = username;
    this.password = password;

    // Here is the coupling I'm talking about
    userAddress = new UserAddress(this.username);
}

public String getUsername()
{
    return username;
}

public void setUsername(String username)
{
    this.username = username;
}

public String getPassword()
{
    return password;
}

public void setPassword(String password)
{
    this.password = password;
}

public String getAddress ()
{
    return this.userAddress.getAddress();
}

@Override
public String toString()
{
    return "username: " + this.username + ", password: " + this.password;
}
}

```

Here we see that the `UserAddress` object is instantiated inside of the `User` object. Also, the `User` has a method that returns its address. Now let's imagine that a developer decides that they want to change the implementation of `UserAddress`, and they don't want to have a `getAddress()` method anymore. This will now break the `User` class. This is what coupling is all about, and it's not a good thing! If we change one Class, we don't want to run the risk of breaking a bunch of others without realizing it.

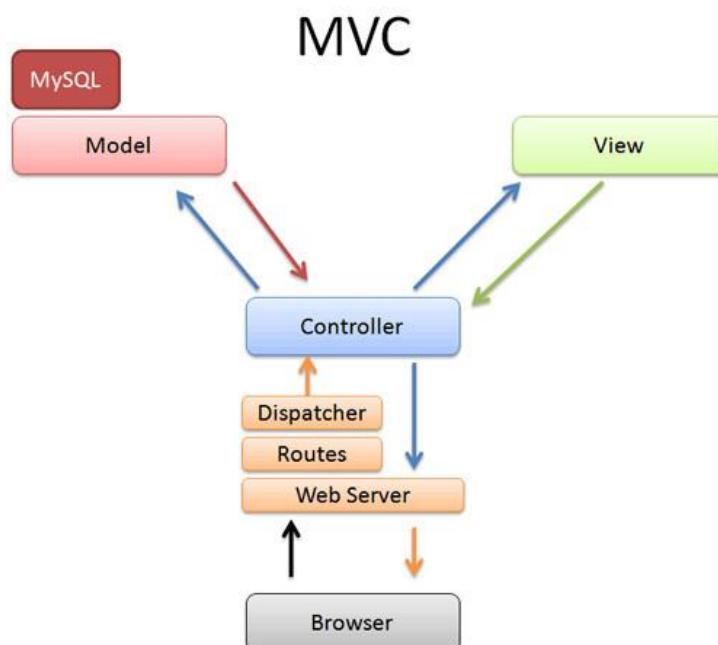
The **Spring Framework** provides a clever solution to ensure that the coupling between Objects is reduced considerably.

What's the MVC design pattern?

As mentioned earlier, the Spring Framework makes use of design patterns (*best practices*). One such pattern is the MVC (model/view/controller) design pattern. This design pattern (*best practice*) is all

about separating your concerns. It's considered good practice to separate the actual user interface code (presentation layer) from the other aspects of your coding (business layer / data layer). When you keep the presentation layer separate from your business layer and data layer, it allows you to "swap out" your presentation layer framework code with another framework (if you so choose). The same goes for keeping your data layer code separate from the other layers, it allows you to more easily swap out the "back-end" of your application if the need arises. I have personally had to do both of these "swaps" before, so I can attest to this "layering" approach being a good practice.

In the Spring Framework, you will see the use of Controllers, these are part of the presentation layer and act as the middle-man between the view and the model.



Hibernate

Hibernate is a framework technology that takes (most of) the need to know SQL out of the equation. It allows you to design your *entire* data layer in terms of Java objects with the use of annotations. Though it's not the easiest framework to use, the internet is bursting with tutorials on the subject of Hibernate.

Here is one that I used when I learned this framework:

http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html_single/

Learning the ins and outs of Hibernate is not required, but it will help you in the long run if you wish to create web applications as part of your career! The examples you see in this eBook with hibernate will be well documented, so no worries!

MySQL

This is the Relational Database Management System (RDBMS) that we will use as our database. It's a free RDBMS and it's very popular, which also means it's well supported and documented on the internet, just like Hibernate. Refer to **Chapter 6** of the "[How to Program with Java](#)" eBook to see details on how to install this on your computer.

HTML

HTML stands for HyperText Markup Language and this is the standard by which webpages are created. One thing you should remember is that HTML is not actually a programming language by any means; it's simply a syntactical structure of text that browsers read and then spit out a webpage. HTML is nice in the sense that it's easy to learn and doesn't require an IDE, you can create a website using just Notepad if you so desire. Here's a quick example of an HTML page:

```
<html>
  <head>
    <title>This is the title - it appears in your browser's tab</title>
  </head>
  <body>
    This is the actual content of the website. This text will just appear as standard black
    text on a white background.
  </body>
</html>
```

That's it ladies and gentlemen. You just copy/paste that code into any file (so long as it's a '.html' file) and you'll be able to open it with a web browser. Simple right?

JavaScript

JavaScript can be thought of the spice that is used in HTML to give it some flavor. JavaScript (a.k.a. JS) is what is used to make HTML websites more interactive. JS can be used to show alert boxes to users on webpages, it can be used to change the content of a webpage dynamically. Let's say for instance we want to display a greeting to a user when they navigate to our website. We can use JavaScript to prompt the user for their name, and then change the webpage's title to say "Hello <your name>", where <your name> is replaced with the actual name that they type in.

JavaScript is often used as the first “line of defense” when validating common user inputs, such as filling out a registration form for a website (ie. First name, last name, email address etc.). JavaScript can check and make sure everything is in order before we actually send all that information off to our database!

JSPs (JavaServer Pages)

The best way to think of JSPs is that it's just HTML (and perhaps JavaScript) with the added luxury of being able to **handle regular Java code** as well. JSPs allow us to place Java code into our webpage view; this is especially handy when combined with the Spring Framework. The Spring Framework has some handy features called tag libraries that allow us to quickly and easily bring data from our view to our controllers and back. Remember that the View is the actual webpage, and the Controller is the middle-man between the View and the Model (data). So, being able to talk back and forth between our Controller and our View definitely comes in handy! Without JSPs helping us out by allowing us to use real Java code, we would be out of luck.

Bonus Content Chapter 2

The Software

Getting Started

Now that you've got some idea of what components make up a web application, let's make sure you've got all the tools you will need to actually create your first web application!

Install Springsource Tool Suite and JDK

For nice clean up to date instructions on how to get your software properly installed, please navigate to the following URL:

<http://howtoprogramwithjava.com/the-5-basic-concepts-of-any-programming-language-concept-5/>

This link will bring you to my blog where I explain step by step how to install the Java Development Kit as well as the IDE you'll need to use to create a web application (Springsource Tool Suite).

If you run into troubles with the installation process, take a look at the comments section of the webpage, as there are good questions and answers to common problems. If you are still experiencing problems just post a comment on that page and either myself or someone from my community will be happy to answer your question. If all else fails, I invite you to email me at trevor@javavideotutorials.net and I should be able to get back to you ASAP.

Install MySQL Server and TOAD

Step 1: Add database library to the classpath

Before we can get started with code, we need to make sure that we have the appropriate library files on the classpath of our application. This is needed in order for Java to understand how to properly “talk” with the database. In this example, we'll be using the MySQL database, so you can grab a copy of their library (JAR) file via this link:

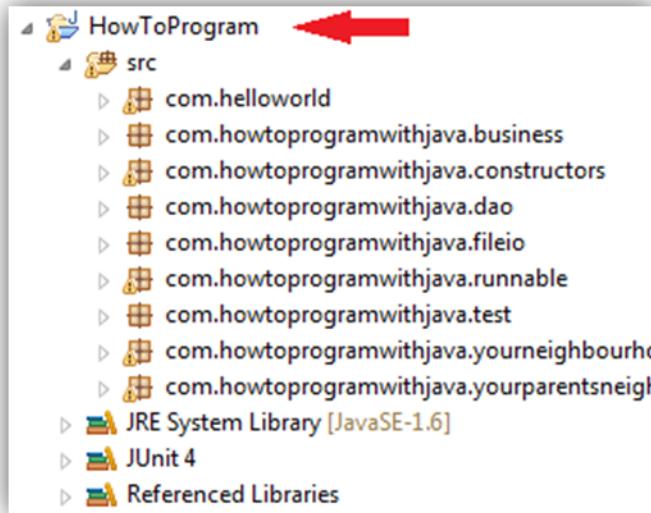
<http://www.mysql.com/downloads/connector/j/>

- When this eBook was published, the MySQL Connector version 5.1.22 was available, so just choose to download the ZIP version.
- When I clicked to download the file, it asked me to login, but there's a link that says “**No thanks, just start my download!**”, this will get the file into your hands ASAP.

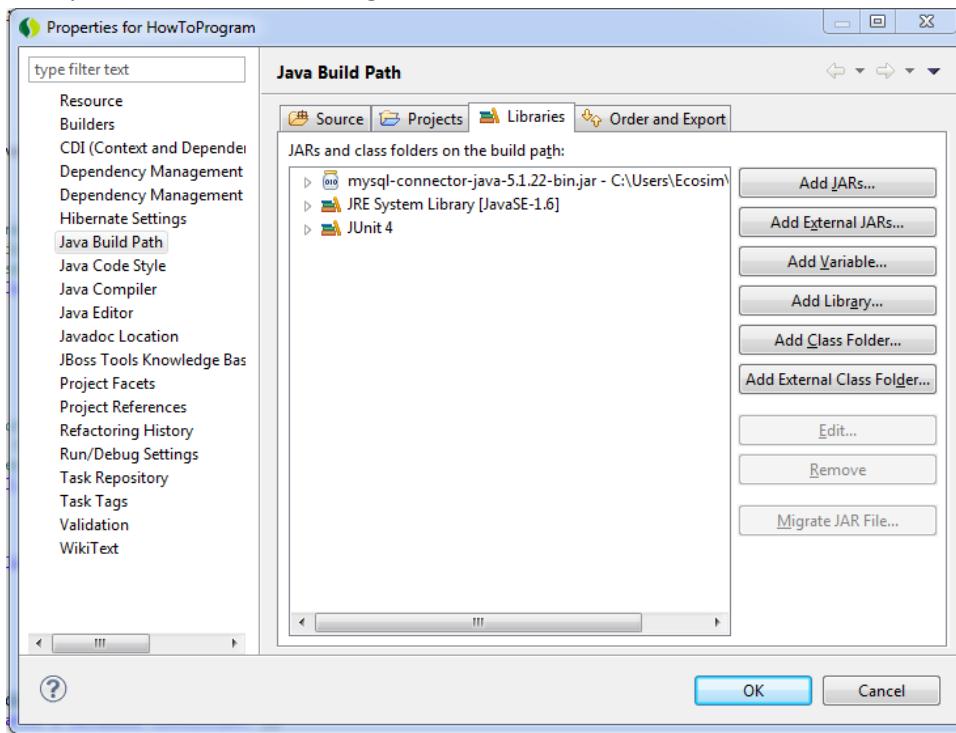
Once you've downloaded the ZIP file, open it up (I use [7-Zip](#)) extract the JAR file to a directory of your choosing (I prefer to keep it with my actual Java application files, but it's up to you). For the record, the JAR file you should be looking for should look something like this: mysql-connector-java-5.1.22-bin.jar.

Okay, we're nearly there, we HAVE the file we need, now we just need to tell our Java application where to find it. This will mean we need to add this JAR file to the application's classpath. To do that, you must:

- Go to your Springsource ToolSuite (STS), right-click on the **project** that you've been working in (in the screenshot below, it's the “HowToProgram” folder). Here's what my **project** structure looks like:



- Choose **properties**
- On the left hand side of the window that appears, click **Java Build Path**.
- In the **Libraries** tab, click **Add External JARs**.
- Choose the **JAR** file that you downloaded and extracted.
- Now you should see something like this:



Excellent, if you've made it this far, then you're in great shape. That was the annoying setup part that's associated with databases, you should really only have to do that whole process once.

Step 2: Download and Install the MySQL RDBMS

This process is a little more involved than just grabbing a file and unzipping it. So I've created a video tutorial that will help you through this step of the process. Just follow this link:

<http://youtu.be/DQULyZAujww>

Bonus Content Chapter 3

Spring ROO

The Secret Sauce

Okay ladies and gentlemen, **here's my secret** for getting started with a web application. I hate going through the craziness of setting up a brand new web application, as there is a lot of "boilerplate" stuff to remember. What I mean by that is that there's a lot of code configuration and annoying settings to tweak in things like XML files to ensure that your web application is setup properly. So my secret is to use a nifty code generation platform called **Spring ROO**.

What this allows us to do is enter some simple command-line type commands and Spring ROO will build the entire scaffolding of our web application for us! How sweet is that!?

Let me show you how it's done! I would recommend trying to follow along with the examples I outline with your own Springsource Toolsuite. As there's no better way of learning than attempting to do what it is I'm teaching.

I've recorded a helpful video that will explain the important parts:

<http://youtu.be/770A3nX2eoU> (40 Minutes, split into 3 parts)

Once you have completed following along with the above video, I recorded another video that expands on the web application we just created with Spring ROO. This video will teach you how to customize the web application created in Spring ROO.

<http://youtu.be/RucGb40z7JA> (90 minutes)

Appendix

Java Tests

Test #1 - Variables

1. Write code below that would declare an `int` variable called `myInteger` and initialize it with the value 500 in one line.

2. What is the console output of the following code? `System.out.println("2-2");`

- 2-2
- null
- 0
- `java.lang.NullPointerException`

3. If you were going to instantiate an integer variable with the name '`i`' and assign a value of `19`, what code would you use?

- `int i = 19`
- `integer i = 19;`
- `i int = 19;`
- `int i = 19;`

4. If you were going to instantiate a `String` variable with the name '`myName`' and assign a value of `Trevor Page`, what code would you use?

- `myName string = Trevor Page;`
- `string myName = Trevor Page;`
- `String myname = "Trevor Page";`
- `String myName = "Trevor Page";`

5. What is the default value for the `String` variable type?

- `""`
- `0`
- `null`
- `1`

6. What is the console output of the following code? `System.out.println(2-2);`

- `2-2`
- `null`
- `0`
- `java.lang.NullPointerException`

7. What is the default value for the `int` variable type?

- `-1`
- `0`
- `1`
- `null`

8. If you need to store this data: `-498.09813` what variable type would you use?

- `String`
- `double`
- `int`
- `string`

9. If you need to store this data: Trevor Page what variable type would you use?

- String
- int
- double
- string

10. Given that you want to store the value 382.28, write the code below to declare a variable called myValue and assign the given value to the variable (in one line)

11. Please select which statement below is false:

- You cannot start a variable name with an underscore.
- You do not need to use the 'new' keyword to instantiate literals
- 823L represents the value 823 of a 'Long' variable
- It is always necessary to assign a value when a field is declared

12. Given that there's a String declared as message, write the code that will output this variable to the console.

13. If you need to store this data: 58 what variable type would you use?

- String
- double
- int
- float

14. Given that you want to store the value "This is a message", write the code below to declare a variable called `message` and assign the given value to the variable (in one line)

Test #2 - Control Structures

1. Write the code to create a `for` loop that will iterate from 0 to 9 (10 iterations) times with the `int i` variable.

2. In what situation would you use a `while` loop?

- When there's a defined start and end
- When there's no defined start and end
- When you only want to execute the code once
- Never

3. How do you combine conditions in a control structure? (i.e. `age` is less than 20 and greater than 12)

- `age < 20 AND > 12`
- `age < 20 and > 12`
- `age < 20 && > 12`
- `age < 20 && age > 12`

4. Write the code to create a `for` loop that will iterate 10 times with the `int i` variable, starting at 3 and incrementing by 2 with every step.

5. When does the code inside a control structure get executed?

- When the program is first executed
- When the code flow evaluates the control structure
- When the condition inside the round brackets evaluates to false
- When the condition inside the round brackets evaluates to true

6. How do you output a blank line onto the console?

- `system.out.println();`
- `System.out.print()`
- `printBlankLine();`
- `System.out.println();`

7. Which of the following is not a valid control structure in Java?

- `if ()`
- `when ()`
- `for ()`
- `while ()`

8. Create a `while` loop that will keep looping `while age is greater than 13 and age is less than 20.`

9. What is the purpose of a control structure?

- To modify the flow of code in a program
- To modify the value of variables in the program
- To organize data in a more efficient manner
- There is no purpose, it's a made up term that means nothing in Java!

10. Write the code for a `while` loop that keeps looping so long as `age > 19`

11. In what situation would you use a `for` loop?

- When there's a defined start and end
- When there's no defined start and end
- When you only want to execute the code once
- Never

12. How would you verify if someone's `age` is 20?

- `if (age = 20)`
- `if (age == 20)`
- `if (age === 20)`
- `if (age equals(20))`

13. What variable type would I use to store this data: `"My birthday is January 1st"`

- int
- double
- char
- String

14. How does code flow in Java?

- From bottom to top and right to left
- From top to bottom and right to left
- From bottom to top and (for the most part) from left to right
- From top to bottom and (for the most part) from left to right

15. Write code below that would ask if `age` is greater than 19.

Test #3 - Control Structures Part II

1. How many times would this code execute the `System.out.println()` statement in this `for` loop?

```
for (int i = 0; i < 28; i++)
{
    for (int j = 0; j < 18; j++)
    {
        System.out.println("inner loop");
    }
}
```

- 18
- 28
- 0
- 504

2. Instantiate a `Date` variable named `theDate` set to the current date/time.

3. Create a nested `for` loop. The outer loop should iterate from 0 to 9 (10 iterations) using variable `i`. The inner loop should iterate from 0 to 14 (15 iterations) using variable `j`

4. Given this code: `while (true) {}` - Choose which statement is the most correct:

- This code is syntactically correct but will not run
- This code is syntactically incorrect and will not run
- This code is syntactically correct but it will run forever
- This code is syntactically correct and is the foundation for the while loop

5. Instantiate a `Calendar` variable named `cal` and assign/set its date to `March 3rd, 2013`

6. How many times would this code execute the `System.out.println()` statement in this `for` loop?

```
for (int i = 0; i <= 28; i++)  
{  
    for (int j = 0; j <= 18; j++)  
    {  
        System.out.println("inner loop");  
    }  
}
```

- 551
- 28
- 504
- 18

7. Which of the following is syntactically incorrect?

- for (int i<0; i=10; i++)
- for (char c='a'; c<'f'; c++)
- for (int j=10; j<25; j = j + 5)
- for (int k=1; k<=10; k++)

8. Create an outer `while` loop that will iterate while `age` is less than 20, and create an inner `for` loop that will iterate 5 times with variable `int i` from 0 to 4

9. Which of the following statements is `true`?

- The while loop keeps looping until the condition is evaluated to true
- The while loop keeps looping until the condition is evaluated to false
- The while loop only iterates a certain number of times, then it stops
- The while loop is always guaranteed to terminate

10. What is the correct order of the three expressions in a `for` loop?

- increment; start; end
- start; end; increment
- end; start; increment
- end; increment; start

11. How many times will this `for` loop iterate?

```
for (int i = 5; i <= 100; i = i + 15)
{
    System.out.println("");
}
```

- 6
- 7
- 8
- 9

12. Which of the following `for` loops is not valid?

- `for (int i=0; i<100; i++)`
- `for (j++; j<10; int j=1)`
- `for (char i='a'; i<'f'; i++)`
- `for (int k=10; k>0; k--)`

13. What is the resulting value of the `date` variable? `Date date = new Date();`

- Today's date, set to the beginning of the day
- Today's date, set to the end of the day
- Today's date, set to the current hour/minute/second
- The first date of the Java's Calendar

14. What is the resulting value of the `date` variable?

```
Calendar cal = Calendar.getInstance();
cal.set(2012, 11, 25);
Date date = cal.getTime();
```

- November 25th, 2012
- November 26th, 2012
- December 25th, 2012
- December 26th, 2012

Test #4 - Data Structures

1. When using the `ArrayList integerArrayList`, how would you retrieve the first item?

- `integerArrayList[1];`
- `integerArrayList[0];`
- `integerArrayList.get(1);`
- `integerArrayList.get(0);`

2. Given that you have an instantiated `ArrayList` called `myList` that stores `Strings`, how would you put a new `String` with value "My String" into `myList`?

3. What code is used to instantiate a `HashMap` with a `String` as the Key and a `List` of `Strings` as the value?

- `Map theMap = new Map;`
- `Map<String> theMap = new HashMap<List<String>>;`
- `Map<String, List<String>> theMap = new HashMap<String, List<String>>;`
- `Map<String, List<String>> theMap = new HashMap<String, List<String>>();`

4. Write an `if` statement that will evaluate to `true` if an `ArrayList` called `myList` has data in it.

5. Which statement is most correct with respect to the difference between `Arrays` and `ArrayLists`?

- An Array's size cannot be changed once it's set
- An `ArrayList`'s size can be changed once it's created
- An `ArrayList` can be instantiated with a suggested size
- All of the above

6. Write the code to retrieve the first element from an `ArrayList` called `myList`

7. What is the purpose of a Data Structure?

- To protect the data from getting lost
- To store data in an organized and efficient manner
- To store data in a way that take up less memory
- All of the above

8. What is the `Map` data structure most useful for?

- Storing Lists
- Storing Arrays
- Storing Key/Value pairs
- Storing Value/Key pairs

9. Given the fully populated `Array intArray` with 10 items, what would happen if you executed this code: `intArray[10]`?

- You would retrieve the 10th element in the Array
- Java would throw an `ArrayIndexOutOfBoundsException`
- You would retrieve the 11th element in the Array
- Nothing would happen

10. Write the code to instantiate an `ArrayList` called `myList` that will store `Integers`.

11. Write the code to instantiate a `HashMap` called `myMap` that will store an `Integer` as its key and a `String` as its value.

12. If I wanted to declare an `Array` of `ints`, what would that look like?

- `[5]int array;`
- `int[] array;`
- `int array[5];`
- `int[5] array;`

13. Given you have the `ArrayList integerArrayList`, how do you insert new objects into this `ArrayList`?

- `integerArrayList.add(1);`
- `integerArrayList[1];`
- `integerArrayList.put(1);`
- `integerArrayList(put);`

14. Given that we've declared an `Array (int [] intArray)`, but haven't instantiated it, what code would we need to write before we could use the `Array`?

- `intArray[3];`
- `intArray = new int[];`
- `intArray[3] = new int[];`
- `intArray = new int[3];`

15. Given the fully populated `Array intArray` with 10 items, how would you retrieve the 4th item in the `Array`?

- `intArray[3]`
- `intArray[4]`
- `intArray.get(3)`
- `intArray.get(4)`

Test #5 - Primitives

1. How could the number `239` be assigned to an `int`

- `int anInt = 239;`
- `int anInt = 0xEF;`
- `int anInt2 = (Integer)239;`
- All of the above

2. What is the default value of the `long` primitive type?

- `0.0`
- `0L`
- `0i`
- `0.0d`

3. Which of the following variable types is not a primitive type?

- `int`
- `float`
- `Boolean`
- `double`

4. Write the code to declare a primitive BOOLEAN called `myBoolean` with the value `false`

5. What is the default value for a `double` primitive instance variable?

- 0
- 0.0d
- 0.0f
- 0L

6. Write the code to declare a primitive DOUBLE called `myDouble` with the value `1.0`

7. Write the code to declare a primitive INTEGER called `myInteger` with the value `-50`

8. If you do not initialize an `Integer` data type, what is the default value assigned to it?

- null
- nil
- 0
- 0i

9. Which of the following statements is true?

- Primitives are more efficient than their Object wrapper counterparts
- Primitives are less efficient than their Object wrapper counterparts
- Primitives are always assigned a default value
- None of the above

10. Given that you have a non-primitive INTEGER named `myInteger`, and you need to assign its value to a `String` called `myString`, how would you accomplish this in one line of code?

11. Which of the following is invalid code?

- `float myFloat = 234.817182304981d;`
- `double myDouble = 465.9823824f;`
- `float myFloat = -823.3f;`
- `double myDouble = -283.1823;`

12. How would you get the `String` equivalent of an `Integer`?

- `Integer.parseInt(myString);`
- `myString.getInteger(myInteger);`
- `myInteger.toString();`
- `Integer.getString(myInteger);`

13. Which of the following data types would not allowed to be used in an `ArrayList`?

- Integer
- double
- Double
- String

14. Write the code to declare a primitive FLOAT called `myFloat` with the value `549.54`

Test #6 - Methods

1. Given the following code, how would make use of the `boolean return` value?

```
public boolean isValidUsername(String username)
{
    if ("tpage".equals(username))
        return true;
    else
        return false;
}
```

- `boolean isUserTpage = isValidUsername("tpage");`
- `isValidUsername("tpage") = boolean isUserTpage;`
- `isUserTpage == isValidUsername("tpage");`
- `boolean isValidUsername("tpage");`

2. What are the four visibility modifiers used in Java?

- private, semi-private, public, semi-public
- public, non-public, private, non-private
- private, protected, package, non-public
- private, package, protected, public

3. Create a `public` setter method (with an empty body) named `setUsername` which takes one `String` argument (`username`) and returns nothing.

4. Create a `public` method (with an empty body) that returns a `boolean` called `isValidData` that takes two arguments (`String username, Integer count`)

5. Why would you use methods?

- It is best to keep all of the programming code within one method for readability
- Using methods allows us to re-use code, thus allowing for robust code
- Using methods tends to make the code less readable
- You shouldn't use methods

6. Write the code for the Java `main` method (the method that flags the starting point of a program in Java).

7. How would you create a method that takes NO parameters, returns NOTHING, with the name `myMethod`

- `public static int myMethod()`
- `public static int myMethod(String firstParameter)`
- `public static void myMethod()`
- `public static void myMethod(String firstParameter)`

8. When using the keyword `private` to create a method, what is the visibility of that method?

- The method is visible to any class other than the one it was created in
- The method is visible only to the class it was created in
- The method is visible to any classes within its declared package
- The method won't be visible to any classes

9. Which of the following is not a valid method declaration?

- `private static void boolean myMethod()`
- `public void myMethod()`
- `private static String myMethod(int param)`
- `private boolean myMethod(Integer param)`

10. When looking at the declaration of the `main` method, what do you now know about it?

- It is a method that returns a boolean and takes a parameter
- It is a method that returns nothing, and takes a String parameter
- It is a method that returns nothing and takes no parameters
- It is a method that returns nothing and takes an array of Strings as a parameter

11. Create a `public` method (with an empty body) named `getUsername` which takes no arguments and returns a `String`

12. How would you create a method that takes a `String` parameter, returns a `boolean`, with the name `myMethod`

- `public static void myMethod()`
- `public static boolean myMethod()`
- `private static boolean myMethod()`
- `private static boolean myMethod(String username)`

13. How would you declare a `public` method that takes a `String` and an `Integer` as parameters?

- `String public myMethod(Integer param)`
- `public void myMethod(String param, Integer param2)`
- `public void myMethod(String param, Integer param)`
- `public String myMethod{Integer param}`

14. When using the keyword `public` to create a method, what is the visibility of that method?

- The method is visible to any class other than the one it was created in
- The method is visible only to the class it was created in
- The method is visible to any classes within its declared package
- The method will be visible to any class

Test #7 - Objects and static keyword

1. Given the object `User`, how would you make a call to the `static` method `validateUser` (which takes no parameters and is declared within `User`)?

2. Write the code to create the class `User` (assuming you don't need imports, package declaration or body)

3. Which of the following statements is not correct?

- You can access a non-static method from a static method
- You can access a static method from a non-static method
- You can access a static variable in a non-static fashion
- You can access a static variable in a static fashion

4. What is a `static` variable?

- A variable defined in a class for which each Object has a separate copy
- A variable defined in a class for which each Object has the same copy
- A variable that can only be assigned certain variable types
- A variable that exists only for a specific amount of time

5. Why do we use getter and setter methods?

- Allows us to encapsulate behaviour
- Allows for insertion of validation when retrieving and modifying values of variables
- Allows us to inject some flexibility into retrieving and modifying values of variables
- All of the above

6. When we declared the `User` Object, why are we able to access methods that we didn't create (like the `toString()` method)?

- User extends `java.lang.Object` functionality
- User is a super class
- User uses static methods
- User uses instance variables and static methods

7. How would you create a `static String` instance variable named `myString`?

8. What in Java is an `Object`?

- Everything
- Everything except static variables
- Everything except primitives
- Everything except instance variables

9. Write the code (according to Java conventions) needed to create the setter method for the instance variable: `private String username`

10. What is a Java Class?

- A place where students learn
- The blueprint for an Object
- The Object of the Blueprint
- A static representation of a variable

11. Given that you had the Class `User` and the `static` variable `securityLevel`, how would you reference the variable from another `Object`?

- `new User().securityLevel;`
- `User().securityLevel;`
- `User.securityLevel;`
- `securityLevel();`

12. Write the code (according to Java conventions) needed to create the getter method for the instance variable: `private String username`

13. What is an instance variable?

- A variable defined in a class for which each Object has a separate copy
- A variable defined in a class for which each Object has the same copy
- A variable that can only be assigned certain variable types
- A variable that exists only for a specific amount of time

14. When reading requirements and deciding what `Objects` to create, what should you look at?

- Verbs
- Nouns
- Pro-Nouns
- Adjectives

15. By default, what is an `Object`'s super class?

- `java.lang.Integer`
- `java.lang.String`
- `java.lang.Objects`
- `java.lang.Object`

Test #8 - Inheritance

1. What main benefit does the `abstract` class have over the `interface`?

- The abstract class is used as a contract
- The abstract class allows you to re-use code
- You don't define the body of any methods
- There is no benefit

2. Write the code to define a `Vehicle` interface that doesn't define any methods. (Assuming no need for imports or package definition)

3. What main benefit does the `interface` have over the `abstract` class?

- The interface allows for code re-use
- You don't need to define the body of some methods
- The interface allows for multiple inheritance
- There is no benefit

4. Which of the following hierarchy chains is invalid?

- Object -> Vehicle -> Car
- Object -> Car
- Object -> Vehicle -> Car -> Vehicle
- Object -> Vehicle -> Bus -> SchoolBus

5. Write the code to define the class `SportsCar` which is a sub-class of the `Car` class. (No need for imports, package declaration or body)

6. When you use an `abstract` class, which of the following statements is `true`?

- You must use the keyword 'extends'
- You must use the keyword 'implements'
- You can extend multiple abstract classes
- You can implement multiple abstract classes

7. When you override the `equals` method, it's recommended that you also override which method?

- `toString()`
- `wait()`
- `notify()`
- `hashCode()`

8. When you use an `interface`, which of the following statements is `true`?

- You must use the keyword 'extends'
- You must implement all public methods
- You must implement all abstract methods
- You must use a child class

9. Which of the following statements is true?

- You can implement as many interfaces as you wish, but you can only extend one class
- You can extend any number of classes you wish, but can only implement one interface
- You must override every method in an abstract class
- All of the above

10. Which annotation do you use when you wish to implement a method in a child Object that is defined in a parent Object?

- @Overload
- @Overthere
- @Override
- @Overunder

11. If a super class defines a method, and a child class overrides this method but wishes to also use the parent class's functionality, what keyword would you use?

- parent
- super
- child
- class

12. In order to ensure that Java doesn't just compare memory locations when comparing two objects, which method should you implement?

- hashCode()
- override()
- overload()
- equals()

13. Write the code which defines the class Person that is a sub-class of the User abstract class. (No need for imports, package declaration or body)

14. Write the code to create the Car class, which contracts the use of the Vehicle interface. (No need for imports, package declaration or body)

15. How would you create an abstract User class? (Assuming you don't need any imports, package declarations or body)

Test #9 - Exceptions

1. When you place code in the `finally` block, the code should not be time sensitive, why is this?
 - It is not guaranteed that the finally block will execute exactly when the try block has executed
 - It is not guaranteed that the finally block will execute at all when the try block has executed
 - This can hog system resources and adversely affect the code outside the try/catch block
 - This statement is false, you can and should place time sensitive code in the finally block

2. What is the `finally` block used for?
 - Executing housekeeping type code (ie. cleaning up database connections)
 - A secondary means to handle thrown exceptions
 - A backup flow of code execution if an error is thrown
 - All of the above

3. If there is NO error thrown in a `try/catch` block, how does the code flow?
 - Into both the catch and finally block
 - Into the catch block only (if one exists) and then continue outside the try/catch
 - Into the finally block only (if one exists) and then continue outside the try/catch
 - It will skip both the catch and finally block

4. There are two keywords that have to do with handling `Exceptions`, what are they?

- try, fail
- throw, catch
- lob, catch
- fail, throw

5. Create a standard `try/catch/finally` block of code that will catch `FileNotFoundException e`, and close a `FileInputStream` named `inputStream` in the appropriate location.

6. Create a standard `try/catch` block that will catch `Exception e`. (no code needed in the body of the try or catch blocks)

7. If you wanted to handle two types of exceptions in one `try/catch` block, how would you go about doing this?

- You cannot handle two separate types of exceptions in one try/catch block
- Declare two catch blocks, each defining which exception to catch
- Pass two exceptions as parameters in the catch block
- None of the above

8. The `java.lang.OutOfMemoryError` is an example of what kind of exception?

- Checked
- Unchecked
- Compile
- Failed

9. The `java.io.FileNotFoundException` is an example of what kind of exception?

- Checked
- Unchecked
- Compile
- Failed

10. How would you define a `public` method named `validatePassword` which throws the generic `Exception` type? (Given that the method returns nothing and takes a `String password` argument)

11. Why is it recommended to declare your variables outside the `try/catch` block?

- Your variables won't otherwise be accessible in the catch and finally blocks
- It's more aesthetically pleasing
- Your variables won't otherwise be garbage collected
- Variables cannot be declared inside a try/catch block

12. `Exceptions` can be broken down into two categories, what are they?

- Passed and Unpassed exceptions
- Thrown and Caught exceptions
- Checked and Unchecked exceptions
- Runtime and Compile-time exceptions

13. If there is an error thrown in a `try/catch` block, how does the code flow?

- Into the finally block only, the code will skip the catch since there was an exception
- Into the catch block only, the code will skip the finally since there was an exception
- Into the finally block if one exists, then into the catch block that matches the thrown exception
- Into the catch block that matches the thrown exception, then into a finally block if one exists

Test #10 - String Manipulation

1. Given `String myString = "This is a test sentence."`, what `String` is returned when we invoke `myString.replace("te", "")`?

- "This is a st sentence."
- "This is a st sennce."
- "This is a test sennce."
- "This is a test sentence."

2. What method is used to 'truncate' a part of a `String`

- `removeString()`
- `truncate()`
- `substring()`
- `parseString()`

3. Given the `String "This is a test sentence."`, when using `indexOf("test")` what is the resulting value?

- 9
- 10
- 11
- 1

4. Given two strings (`String myString, String mySecondString`) write the code that would concatenate these two strings together and assign the result to another variable called `result` (don't forget to define the `result` variable)

5. Write the code that would output a message to the console stating "Your name is " and then concatenate the variable `name` to this message.

6. How do you remove whitespace (leading and trailing white space) from a `String`?

- `trim()`
- `replace()`
- `removeWhitespace()`
- `shrink()`

7. Given the `String` "This is a test sentence.", when using `indexOf("tests")` what is the resulting value?

- 9
- 10
- 11
- 1

8. What is the best way to compare two `Strings`

- `string1.equals(string2);`
- `string1 == string2`
- `string1 = string2`
- `"string1" == string2`

9. Write the code to fabricate an `if` statement that will check if the `myString` variable contains the word `"fox"`. (no code needed in the body of the if statement)

10. How do you place a new line into a `String`?

- `\t`
- `\n`
- `\t`
- `\n`

11. If we needed to change all occurrences of a particular `String` within another `String`, what method would we use?

- `replace()`
- `searchReplace()`
- `findReplace()`
- `restore()`

12. If you have a `User` object defined and wish to print the contents to the console in a human readable way, what must you do?

- `System.out.println("User")`
- Use concatenation with a second `User` object
- Overload the `toString()` method of the `User` object
- Override the `toString()` method of the `User` object

13. The process of 'mashing' multiple `Strings` together is known as?

- String manipulation
- String overriding
- String concatenation
- String overloading

14. Write the code to assign the following `String` literal to a `String` variable called `myString`:

```
"This is a sentence  
with a line break inside of it!"
```

Test #11 - Overriding and Overloading

1. When overriding a method, it is advisable to use the _____ annotation if you're using Java 1.6+

- @Overload
- @override
- @Override
- @Overide

2. Let's say you're given an `int`, but your code requires a `double`, what must you do to appropriately convert your value?

- convert
- deploy
- cast
- transform

3. Overriding can only happen when you have a _____ and _____ class

- parent, super
- super, child
- super, private
- private, public

4. When is it always legal to cast? ie. when you are guaranteed not to get a `ClassCastException`

- rightcast
- downcast
- leftcast
- upcast

5. What is the difference between Overriding and Overloading?

- Overriding has implications for Inheritance, Overloading has implications for Polymorphism
- Overriding is part of Java 1.5, but Overloading was introduced in Java 1.1
- Overriding replaces parent functionality, Overloading supplies multiple implementations for a given method name
- Overriding allows you to change behaviour at compile time, Overloading is done at runtime

6. What is the main downside to casting?

- You could get an Exception
- You could accidentally modify the value of your variable
- You could cause an inability to get an `equals()` match on your objects
- None of the Above

7. Given the variable `Object myString`, how would you define a `String anotherString` variable and assign `myString` to it?

8. Given the method `boolean isAnAnagram(String word1, String word2)`, how would you overload this method?

- `String isAnagram(String word1, String word2)`
- `boolean isAnagram(char[] word1, char[] word2)`
- `String isAnAnagram(String word1, String word2)`
- `boolean isAnAnagram(char[] word1, char[] word2)`

9. Overloading allows us to define two or more methods within the same _____ that share the same _____

- Class, name
- method, name
- package, name
- package, Class

10. The compiler does not consider _____ when differentiating methods

- arguments
- return types
- method names
- method signatures

11. When overriding a method, both the _____ and the _____ have to match

- method name, parameters
- method name, return type
- return type, parameters
- return type, classes

Test #12 - Collections Part I

1. In order to insert an element into a `List`, what method do you use?

- `insert()`
- `add()`
- `pop()`
- `put()`

2. Write the code that is needed to instantiate an `ArrayList` called `myArrayList` which stores `Strings` (remember to program to an `interface`)

3. A `List` is a(n) _____ and an `ArrayList` is the _____ of a `List`.

- interface, implementation
- implementation, interface
- object, counterpart
- collection, counterpart

4. What's a compelling reason why someone would use a `LinkedList`?

- It has better performance than an `ArrayList`
- It is easier to use than an `ArrayList`
- Because it's an index based List
- It can mimic a Queue

5. How do you delete the second element in an `ArrayList` called `myArrayList`?

6. What method is used to remove an element from a `List`?

- `remove()`
- `pull()`
- `delete()`
- `destroy()`

7. With respect to multi-threading, the `ArrayList` is _____ and the `LinkedList` is _____.

- not synchronized, synchronized
- not synchronized, not synchronized
- synchronized, not synchronized
- synchronized, synchronized

8. In order to properly check if a `List` is empty, you must do the following:

- `if (list == null && list.isEmpty())`
- `if (list != null && list.isEmpty())`
- `if (list == null || list.isEmpty())`
- `if (list != null || list.isEmpty())`

9. What is the main reason why inserting elements into the middle of a `LinkedList` is faster than with an `ArrayList`?

- The ArrayList is poorly implemented
- The LinkedList just shuffles a couple of indexes around
- The LinkedList just shuffles a couple of pointers around.
- The ArrayList is not able to restructure itself when an element is inserted.

10. An `ArrayList` relies on a(n) _____ to keep track of its elements?

- counter
- index
- hashmap
- reference to next element

11. A `LinkedList` relies on a(n) _____ to keep track of its elements

- counter
- index
- hashmap
- reference to next element

12. Write the code that is needed to instantiate a `LinkedList` called `myLinkedList` which stores `Integers` (remember to program to an `interface`)

13. What's the main difference(s) between the `List` and `Set` collections?

- Lists are synchronized
- Lists store elements
- Lists allow duplicates
- All of the above

14. How do you write a "for each" loop to iterate through an `ArrayList` of `Strings` called `myArrayList` (assigning each element to the variable `item`)

15. Given you have a `LinkedList` called `myLinkedList`, how would you insert the `String` "An element" into it?

Test #13 - Collections Part II

1. Write the code to insert a new key/value pair into `myHashMap`. The key is `100`, the value is `"Trevor Page"`.

2. With the `HashMap` implementation, they map's keys must be _____

- sorted
- Strings
- unique
- numeric

3. Write the code to declare and instantiate a `TreeSet` called `myTreeSet` that stores `Strings`. (Remember to program to an `interface`)

4. Write the code to declare and instantiate a `HashMap` called `myHashMap` that stores an `Integer` for a key and a `String` for a value. (Remember to program to an `interface`)

5. Write the code to declare and instantiate a `HashSet` called `myHashSet` that stores `Integers`.
(Remember to program to an `interface`)

6. Write the code to retrieve the value from `myHashMap` given key `100`.

7. Which of the following `Sets` maintain insertion order?

- `HashSet`
- `LinkedHashSet`
- `TreeSet`
- `HashTableSet`

8. What happens when you add a duplicate value into a `Set`?

- The Set ignores the duplicate value and the add method returns false
- The Set replaces the existing value with the new value and returns true
- The Set allows the duplicate to be inserted, but the add method returns false
- The Set will throw a `DuplicateKey` exception

9. What is the main reason you would use a `Set`?

- To automatically eliminate duplicates
- To always ensure natural ordering of Objects
- To allow for automatic insertion ordering of Objects
- To allow for the storage of key/value pairs

10. Just like the `ArrayList` and `LinkedList`, the default implementation of `Sets` and `HashMap` we looked at _____.

- are always thread safe
- are never thread safe
- are only sometimes thread safe
- are only sometimes not thread safe

11. What happens when you insert a duplicate key into a `HashMap`?

- the `HashMap` will ignore the newly added key/value pair and leave the existing entry
- the `HashMap` will throw a `DuplicateKey` exception
- the `HashMap` inserts a duplicate key/value pair into the bucket
- the `HashMap` replaces the existing key/value pair with the newly added key/value

12. If you would like to insert an `Object` into a `Set`, which method would you use?

- add
- put
- insert
- place

13. Which of the following `Sets` maintains order?

- `HashSet`
- `LinkedHashSet`
- `TreeSet`
- `HashTableSet`

14. If you would like to insert an entry into a `HashMap`, which method would you use?

- `add`
- `insert`
- `place`
- `put`

15. What are the main differences between a `Set` and a `List`?

- `HashSets` maintain order whereas `Lists` do not
- `HashSets` only store Objects that extend the `Set` class
- `HashSets` use the 'put' method whereas `Lists` use the 'add' method
- None of the above

Test #14 - Operators

1. What is the shortcut to incrementing `int j` by 1?

- `j = j + 1;`
- `j++;`
- `j+=;`
- `j=+`

2. If you have the variable `myVariable` that could potentially be `null`, how would you safely invoke its `equals` method?

- `if (myVariable.equals("something"))`
- `if (myVariable != null && myVariable.equals("Something"))`
- `if (myVariable != null & myVariable.equals("Something"))`
- `if (myVariable != null || myVariable.equals("Something"))`

3. What is the proper format for a ternary operator?

- `int condition = (result) ? result1 : result2`
- `int result = (result1) ? condition : result2`
- `int result = (condition) ? result1 : result2;`
- `int result = (result2) ? result : result2`

4. Write the shortcut code for incrementing the `int i` by 1.

5. Write an `if` statement that checks to see if two `ints` (`int1, int2`) are not equal.

6. What is the result of the following condition: `true || false`

- true
- false

7. What is the shortcut to decrementing `int k` by 32?

- `k--;`
- `k = k - 32;`
- `k-=32;`
- `k-32;`

8. How do you compare if the values of two `ints` are equal to each other?

- `a == b;`
- `a = b;`
- `a.equals(b);`
- All of the Above

9. What is the result of the following condition: `true && false`

- true
- false

10. Write the shortcut code for incrementing the `int i` by 10.

11. How would you ask if two `int` values are NOT equal to each other?

- `a <> b`
- `a not b`
- `!a.equals(b)`
- `a != b`

12. Write the code to create an `if` statement checking two `ints` (`int1` and `int2`) for equality.

13. Write the code to create an `if` statement checking two `Integers` (`integer1` and `integer2`) for equality.

14. How do we properly check if someone is a teenager (assuming the values will always be positive `int` values)?

- if (`age > 12 && age < 20`)
- if (`age >= 13 && age <= 19`)
- if (`!((age > 0 && age < 13) || (age > 19))`)
- All of the Above

15. What would be the result of this remainder operation: `52 % 7`?

- 0
- 1
- 2
- 3

Test #15 - This keyword

1. Create a `public` no-argument constructor for the `User` class (with nothing in the body)
2. Given the Class `HumanBeing`, what is the proper way to execute a constructor from inside another constructor?
 - `this.HumanBeing();`
 - `HumanBeing();`
 - `this();`
 - None of the Above
3. Create the setter method (using standard Java conventions) for the instance variable `private String username`
4. When is the `this` keyword commonly used in Java code?
 - When referring to an instance of an Object
 - Inside of the setter method of an Object
 - Inside the constructor of an Object
 - All of the Above

5. The `this` keyword always refers to an instance of an `Object`

- true
- false

6. What happens when you don't qualify one of the variables inside of a typical setter method?

- Nothing happens
- There is a compilation error saying that the assignment has no effect
- You get a runtime error when the assignment occurs
- You get a warning saying that the assignment has no effect

7. Can you use the `this` keyword to get a reference to a `static Object`?

- Yes
- No

8. What's the reason we use the `this` keyword in a typical setter method?

- It is good form to use the 'this' keyword in a setter
- It's actually not a good practice to use the 'this' keyword in a setter
- To avoid a name collision with the parameter
- None of the Above

9. Create a `private` two-argument constructor for the `User` class that takes `String username,` `String password` (with nothing in the body)

Test #1 - Answers

1. `int myInteger=500;`
2. A
3. D
4. D
5. C
6. C
7. B
8. B
9. A
10. `double myValue = 382.28;`
11. D
12. `System.out.println(message);`
13. C
14. `String message="This is a message";`

Test #2 - Answers

1. `for (int i=0; i<10; i++) {}` OR `for (int i=0; i<=9; i++) {}`
2. B
3. D
4. `for (int i=3; i<22; i=i+2) {}` OR `for (int i=3; i<=21; i=i+2) {}`

5. D
6. D
7. B
8. `while (age > 13 && age < 20) {}` OR `while (age >= 14 && age <= 19) {}`
9. A
10. `while (age > 19) {}` OR `while (age >= 20) {}`
11. A
12. B
13. D
14. D
15. `if(age>19) {}` OR `if(age>=20) {}`

Test #3 – Answers

1. D
2. `Date theDate = new Date();`
3. `for (int i=0; i<10; i++) { for (int j=0; j<15; j++) { } }` OR `for (int i=0; i<=9; i++) { for (int j=0; j<=14; j++) { } }`
4. C
5. `Calendar cal = Calendar.getInstance();cal.set(2013, 2, 3);` OR `Calendar cal = Calendar.getInstance();cal.set(2013, 02, 03);`
6. A
7. A

8. `while (age < 20) { for (int i=0; i<5; i++) { } }` OR
`while (age <= 19) { for (int i=0; i<=4; i++) { } }`
9. B
10. B
11. B
12. B
13. C
14. C

Test #4 - Answers

1. D
2. `myList.add("My String");`
3. D
4. `if (!myList.isEmpty()) {}` OR `if (myList != null && !myList.isEmpty()) {}`
5. D
6. `myList.get(0);`
7. B
8. C
9. B
10. `List<Integer> myList = new ArrayList<Integer>();` OR
`ArrayList<Integer> myList = new ArrayList<Integer>();`

11. `Map<Integer, String> myMap = new HashMap<Integer, String>();` OR `HashMap<Integer, String> myMap = new HashMap<Integer, String>();`
12. B
13. A
14. D
15. A

Test #5 – Answers

1. D
2. B
3. C
4. `boolean myBoolean = false;`
5. B
6. `double myDouble = 1.0;` OR `double myDouble = 1.0d;`
7. `int myInteger = -50;`
8. A
9. A
10. `myString = myInteger.toString();`
11. A
12. C
13. B
14. `float myFloat = 549.54;` OR `float myFloat = 549.54F;`

Test #6 – Answers

1. A
2. D
3. `public void setUsername (String username) {}`
4. `public boolean isValidData (String username, Integer count) {}`
5. B
6. `public static void main (String[] args) {}`
7. C
8. B
9. A
10. D
11. `public String getUsername () {}`
12. D
13. B
14. D

Test #7 – Answers

1. `User.validateUser();`
2. `public class User {}`
3. A

4. B
5. D
6. A
7. `private static String myString;`
8. C
9. `public void setUsername(String username)
{this.username = username;}`
10. B
11. C
12. `public String getUsername() {return this.username;}`
13. A
14. B
15. D

Test #8 – Answers

1. B
2. `public interface Vehicle {}`
3. C
4. C
5. `public class SportsCar extends Car {}`
6. A
7. D
8. B

9. A
 10. C
 11. B
 12. D
13. `public class Person extends User {}`
 14. `public class Car implements Vehicle {}`
 15. `public abstract class User {}`

Test #9 – Answers

1. A
 2. A
 3. C
 4. B
5. `try{} catch (FileNotFoundException e) {} finally {inputStream.close();}`
 6. `try{} catch (Exception e) {}`
7. B
 8. B
 9. A
10. `public void validatePassword(String password) throws Exception {}`
11. A
 12. C

13. D

Test #10 - Answers

1. B

2. C

3. B

4. `String result = myString + mySecondString;` OR `String result; result = myString + mySecondString;`

5. `System.out.println("Your name is " + name);` OR
`System.out.print("Your name is " + name);`

6. A

7. D

8. A

9. `if (myString.indexOf("fox") > -1) {}` OR `if (myString.indexOf("fox") != -1) {}`

10. D

11. A

12. D

13. C

14. `myString = "This is a sentence\nwith a line break inside of it!";` OR `String myString = "This is a sentence\nwith a line break inside of it!";`

Test #11 – Answers

1. C
2. C
3. B
4. D
5. C
6. A
7. `String anotherString = (String)myString;` OR `String anotherString; anotherString = (String)myString;`
8. D
9. A
10. B
11. A

Test #12 – Answers

1. B
2. `List<String> myList = new ArrayList<String>();`
3. A
4. D
5. `myArrayList.remove(1);`
6. A
7. B

8. C
 9. C
 10. B
 11. D
- ```
12. List<Integer> myLinkedList = new
 LinkedList<Integer>();
```
13. C
- ```
14. for (String item : myArrayList) {}
```
- ```
15. myLinkedList.add("An element");
```

## Test #13 – Answers

1. myHashMap.put(100, "Trevor Page");
2. C
3. Set<String> myTreeSet = new TreeSet<String>();
4. Map<Integer, String> myHashMap = new HashMap<Integer,  
 String>();
5. Set<Integer> myHashSet = new HashSet<Integer>();
6. myHashMap.get(100);
7. B
8. A
9. A
10. B
11. D

12. A

13. C

14. D

15. D

## Test #14 - Answers

1. B

2. B

3. C

4. `i++;`

5. `if (int1 != int2) {}`

6. A

7. C

8. A

9. B

10. `i += 10;`

11. D

12. `if (int1 == int2) {} OR if (int2 == int1) {}`

13. `if (integer1.equals(integer2)) {} OR if (integer2.equals(integer1)) {}`

14. D

15. D

## Test #15 – Answers

1. `public User () { }`
2. `C`
3. `public void setUsername(String username)  
{this.username = username;}`
4. `D`
5. `A`
6. `D`
7. `B`
8. `C`
9. `private User (String username, String password) {}`