DANIEL KOENIGSBERG

JAVA PRIMER FOR THE IMPATIENT

DANIEL KOENIGSBERG

TUFTE-LATEX.GOOGLECODE.COM

First printing, June 2023

Contents

Dedicated to Frank

A word of warning...

These lessons were written by me several years ago (around 2015) in order to teach my friend how to program in Java and are presented "as is." They may no longer represent up to date information on the state of Java programming and may contain many typographical, grammatical, and content errors. This content has not been edited since its creation in any way.

Hello World

Introduction

An IDE or integrated development environment is a tool which programmers can utilize to effectively program. There is nothing wrong with coding in any text editor you choose, from VIM to IntelliJ. For the purpose of this text I will be making references to Eclipse which I have no doubt will become quickly outdated by changes to the Eclipse UI if it even exists at the time of your reading this. While we could deal with this problem any number of ways, I think the most prudent one is to provide the information as a potential tool and to allow you, the reader, to utilize the insane invention of the Internet to fill in the missing pieces.

This is not to be cruel or lazy on my part but instead to provide you with the exact kind of thing you are likely to encounter out in your travels as a programmer. You will come across many outdated resources, many lone form posts which ask the exact question you are asking but without an answer, and many incorrect-but-mostly-correct solutions. A good programmer knows how to tell the difference and research the problem. At its heart that is what this book is about.

Getting Started

We will begin on our quest of understanding by first installing some form of IDE or text editing program on which to write our code. As I mentioned before you should install Eclipse if you are unfamiliar with development environments or are blindly falling along with this reference (which I do not recommend). You may also need to install Java on your computer and I leave it to you to figure out how to do that.

The first thing you're going to notice is that the user interface for Eclipse is both complicated and frustrating. Later on you'll love it. On the left there should be a thing called "package explorer."

- Right click there.
- New > Java Project
- Now click on Java Project and for project name type *ZerothLesson*
- Click on the arrow to the left of the folder icon to open the folder and then do the same under *src*
- Right click on src,
- New > Package, and name it *lesson*
- Package names are typically one lower case word that describes their purpose. Don't worry about this.
- Right click on your package
- New > Class, type the name *HelloWorld* and make your first class!

It is common practice to name classes using a noun phrase. Capitalize all first letters of words in your class names. This is called UpperCamelCase. lowerCamelCase is when we do not capitalize the first word's first letter. This will be used later.

Typically the subject of classes is weightier but in Java you must always be working in a class. We will not utilize the features of classes yet, but we will need to write our code in the body of one. We will return to this later when we talk about OOP or object oriented programming.

When eclipse makes a new class it saves the code as a .java file with the same name. Always match your class name to your file name. If you decide to change the name of a class, right click on the file in the package explorer and click re-factor name-space or copy paste your code into a new class with the appropriate name if you cannot locate the re-factor option.

Do not simply edit the class name and then try to rename the .java file. I have never attempted this but it seems like a huge waste of time and energy and your friends will probably be embarrassed for you.

What you should see now:

```
public class ZerothClass{

}
```

In Java the public keyword means that other classes in the same package can access the class you are declaring. The class keyword declares a class called ZerothClass. ZerothClass is an object, but we will not use it as such.

The following code snippet will not make any sense to you. Type it, anyway (println is short for print line, the l is a lowercase L not a capital i).

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

Now save your work and right click on the "ZerothClass.java" file in the package explorer. Right click > run as > java application. On the bottom there is Problems, Javadoc, Declaration, Console. Make sure you have console tab selected. Hello World! should have appeared in this box.

Before getting into what we have just done, look at the Outline box on the right. You'll notice it says:

```
HelloWorld
main(String[]): void
```

This is an awesome feature of eclipse. Keep an eye on it as we move forward. Try to figure out what it does on your own.

Discussion

OK now we can talk about what we just did. The first thing to say is that we have declared a method (which you may be familiar with from other programming languages, such as Python, as a function). To declare a method in Java we must use some keywords and they can be pretty confusing. So I'll explain what they mean so far. But first, what is a method?

A method is exactly like a mathematical function. It's a black box that performs an operation. Methods have two very important fields, a return type and an argument(s). If you think of $f(x, y) = cat$ as a method, the arguments are x and y, and the return type is cat (or an object of type, animal). Methods take arguments, perform the code within them, and then spit out the result which will be the same type of object as the return type specified. You will declare methods abstractly i.e. $f(x) = x^2$, not $f(10) = 100$. Methods are used to recycle code conveniently.

Now back to the keywords we have used:

- public (for methods): the method can be used by any class in the same package

- static: the method can only be used within the class.[1]

[1] When the static keyword is used on something it means that thing is unique to the class. All instances of that class will use the same static method. For this reason we do not need to instantiate a member of the class in order to call the function. You can use the static keyword on other things too. The global purpose of the static keyword remains outside of our limited framework for now but it is important to introduce these topics early.

- return type: in this case void: the type of object or primitive that the method will return. Void means nothing is returned. Void is kind of a dummy keyword. Void just tells Java that the method does not actually return something.[2,3]

- main: this is not a keyword, it is the name of the function. Syntactically, we declare methods with: keywords returntype name(//whats being passed)//some code

Now lets unpack all that other shit we did. First, Brackets []: indicate an array of type whatever (in this case of Strings) is written before them. String[] is an array of strings. In Java when we declare what will be passed, we declare the type first and then give it a dummy name (in this case "args")

In Java in order to execute code we need a main method. This is always a public static method which returns void. It takes as an argument an array of strings. Why does it take an array of strings? It is taking as an argument the array of strings in console! This is how it knows to run in the first place.

Read more, but ignore most of: http://csis.pace.edu/ bergin/KarelJava2ed/ch2/javamain.html

[2] This is where a stark contrast forms between f(x) and a method. There are two types of methods **accessors** and **mutators**. Accessors simply access data. Mutators change things. Neither type says anything about the return type, but we rarely think of a function $f(x)$ which takes a value x and changes it to y without us seeing the result.

[3] There is not necessarily a preferred type of method. In **functional programming** mutators are never used. In most other cases it is common practice to use both kinds of methods in the same application.

Executing Code

The main method is necessary to make a .java runnable and must be included in at least one of the .java files in a package. What is typed in the main method will execute when the .java file is run. Just as in other languages, putting anything in quotes means it is a string. Thus the "Hello World!" bit of our code is a string which has not been associated with a variable. Another implementation that works is:

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

or using an out of scope variable, instead:

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

or creating a new string object; with strings this initializes to "".
When you see a += b it means a = a + b.

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

or declare a String, in this case it is not initialized and we must initialize it before use:

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

Further Discussion

What are the brackets({) and shit doing? In Java we put blocks of code in brackets. Unlike python spacing is not sufficient. In fact spacing in Java does nothing. It only helps the reader read the code.[4] For this reason it is not sufficient to hit enter after a line of code either. A line of code must end with a semicolon.

[4] Which is important.

In Java = is the assignment operator. Whatever is left of = will be assigned to whatever is right of =. What are () for? These are used exclusively for methods (and for order of operations in the classical sense i.e. PEMDAS). System.out.println is a method which will take anything for an argument and print it to console. That's right. I can create a class called BonerClass and create an instance of that class called erectBoner and if I type System.out.println(erectBoner); something will happen. What will happen in this case is that it will print the representation of the pointer for erectBoner. This is because Java **passes objects by reference**. If you want to print erectBoner in any other way, you need to explicitly make a toString method for the class, BonerClass.

If you didn't understand all of this, that's ok (in fact, you shouldn't). You should still read the next bit. What does System.out.print do? It prints to console. What is the difference between System.out.println and System.out.print? The former prints a line after printing the argument. The latter does not.

For System.out.print(arg); if we replace arg with ("some shit: " + 7) *someshit: 7* is what will appear on the console. This is because of Java's ability to intuitively concatenate Strings before processing them.

A quick discussion of Strings and Arrays before we move on: An array is list of objects with a fixed size. The way an array is implemented in memory is that when the array is declared (and the size initialized) a portion of contiguous memory is set aside for that array. When things are stored in an array they are stored next to each other, so that getting an object in an array is a very quick and simple process for the CPU. Key thing to take away here: arrays are sequences with a fast look up time.

A String is basically an array, except that it is implemented to serve a different purpose. Strings are stored in contiguous memory but they also support more operations than arrays. Strings are not fixed size (at least not in so far as usage) and generally are used for storing words and messages.

Now you are ready to move on to other shit (arguably you were as soon as you ran the first HelloWorld method).

Variables, Arithmetic, Operators, Recursion, Loops, and Shit

Introduction

Now that you know a little bit about how Java works, it is time to start doing some shit in Java. We will start with variables. Just like Python you can import libraries from other crap. In fact we are importing print(); and println(); from System.out. Pretty cool eh? These are packages given to you by the developers of Java. You may want to import Math in this one. Try to figure out how to import Math on your own. Some IDEs (such as eclipse) are helpful and if you type something that you need to include, it will underline it. Mousing over it should give you the option "Import whatever" which will add code to the preamble of your class. This is not important right now. You should be able to do everything here without importing Math.

Primitives

Primitives are data types. When you want to declare a variable of a primitive the syntax is //primitive type// //variable name// = //value//; There are several types of primitives but we will only use int, long boolean, char, and double for now. Each primitive establishes itself from a concept with which you may already be familiar. In computers data is stored as binary (1s and 0s). We call an individual 1 or 0 a bit. For data types which need to store more than two values (e.g. integers) we have larger data types consisting of multiple bits. The following guide should be helpful in understanding how many bits are associated with the primitive type.

- byte: 8 bits
- short: 2 bytes
- int: 4 bytes (we will largely use these or (sometimes) longs for representation of integer numbers)

- long: 8 bytes
- boolean: essential 1 bit, but really just true(1) or false(0)
- floats, doubles, chars (additional reading for now)[5]

[5] https://docs.oracle.com/javase/tutorial/java/nutsandbo

As a rule, the number of values we can store with n bits is 2^n. Consider the example where we have two bits, b_1b_2 $b_i \in 0, 1$. Then our $2^2 = 4$ possible values are $00, 01, 10, 11$. How we assign these bits is up to us in a strictly abstract sense. For the computer, 00 would represent 0, 01 would represent 1, 10 would represent 2, and 11 would represent 3.

Binary Operators

In all of the following definitions "=" is the math equals not the assignment =. However, anything else, including ==, is explained as a binary operator. A **binary operator** is basically a method which takes as an argument exactly two things. We will return to this idea. We are also only defining these binary operators in terms of integers.

First I would like to introduce you to the mathematical/logical concept of **if and only if**. In order to do this we will need to first touch on the concept of if A then B. We write this statement symbolically as $A \rightarrow B$. In this case $\rightarrow$ is a binary operator acting on two booleans, A and B. Intuitively we can think of this as a function $\rightarrow (A, B)$. But what does the function body look like?

```
public static boolean rightArrow(boolean A, boolean B){
        if (A && !B) return false;
    else return true;
}
```

In other words we consider the statement, "If A then B" to be a true statement as long as B is ALWAYS true when A is true. To think about this intuitively, we cannot claim "if it rains there is water" if there is no water but it is raining. However, if there is no rain the truthiness of the statement remains intact regardless of whether or not there is water. Do you understand what the symbol, "&&" is doing?

WHEN WE SAY **if and only if** we refer to the binary operator characterized by the code

```
public static boolean ifAndOnlyIf(boolean A, boolean B){
        return rightArrow(A, B) && rightArrow(B, A);
}
```

Again we see that same "and" operator. In case you have not figured it out yet, that double ampersand is also a binary operator. We will properly define it below. Intuitively we can think of "if and only if" as something like an equality. When we say "A if and only if B" the values of A and B must match. It is the equivalent of saying "if A then B AND if B then A."

Now that we have a mathematical tool for use in the definitions of our upcoming binary operators we can commence with the ones you are more likely to encounter in the coding environment. We will start with the + operator. For two integers X and Y, $X + Y$ is the additive sum of X and Y. This is probably not surprising. Similarly - is the additive inverse operator, or subtraction operator.

Let us instead define **subtraction** through addition. Let Z be an integer such that $Z + Y = 0$.[6] Then, we can define subtraction as $X + Z = X - Y$. This is why we called - the additive inverse operator. Subtraction is just addition of the additive inverse.

[6] Or Y is the additive inverse of Z

Now let us introduce **multiplication**, *. For integers X and Y, $X * Y$ is equal to X added to itself Y times. If we are gluttons for punishment we can use this reliance on addition to define $X * Y = \sum_1^Y X$. Notice that we can indeed define most common binary operators which we are used to as specific cases of addition.

We define /, the **division** operator for integers X and Y unconventionally. In this case Java will return an integer. For this reason I will express it this way: Let X/Y = Z where Z is also an integer (not to be confused with a decimal). Then, X - Z*Y = A where A is the remainder of dividing X by Y in the conventional sense. When using the division operator with two integer primitives the answer given will be Z. Decimals are not included and intuitively this is sort of like finding the number of times Y goes into X without going over. We will return to a better definition in a second. $1/10 = 0$, $11/10 = 1$. We can avoid this by passing decimals (e.g. $1/10.0 = .1$).

```
// The following is pseudo code meant to be intuitively helpful
public static integer integerDivision(int X, int Y) {

    \\ Division works in the classical sense with doubles
    double x = (double) X;
    double y = (double) Y;
    double Z = x / y;

    \\ Floor is the equivalent of "rounding down" or
    \\ truncating the decimals
    return Math.floor(Z);
}
```

You may be saying "well that's great and all, but how do I get the remainder?" You will find your answer in the **modulo**, % operator. For two integers X and Y, $X\%Y = Z$ where for some integer n, $X - Z = Y * n$. That is to say, $X\%Y$ is the remainder of dividing X by Y. Let $X/Y = C$. Then, $X - C * Y = X\%Y$. Note that it is possible to reconstruct division with remainder using modulo and integer division.

```
public static integer[] divisionWithRemainder(int X, int Y) {
    // First declare a results array to contain the two values
    int[] results = new int[2];

    // In the first position put the value of integer division
    results[0] = X / Y;

    // In the second position put the remainder
    results[1] = X % Y;

    return results;
}
```

When we want to say two primitives are equal we can use the **equals operator**, ==. this is a boolean (equals). For two integers X and Y, X == Y is true IF AND ONLY IF (iff) X and Y are equal. If we want to look for the opposite (i.e. X is not equal to Y) we can use the **unary not operator**, ! in conjunction with the equals operator, X != Y or !(X == Y), whichever you prefer.

Lastly, when we want to "and" two booleans (i.e. we want to check that they are both true), a and b we can use the **binary and operator**, &&. If we want to check that at least one of them is true we can use the **binary or operator**, ||. What if we want to check either a or b? We would use the **exclusive or operator**. In this case we can use the ˆ operator which allows us to check or values on the bits within our primitives (e.g. 00ˆ11 = 11 but 01ˆ11 = 10).

This is a **bitwise** operator, not a **conditional** operator. The former performs operations on bits as if they were booleans, the latter on booleans as we will use them in Java more frequently. As an exercise try to construct your own exclusive or operator using pseudo code.[7]

[7] Hint: you can use the not operator, ! on a boolean to get the opposite value for that boolean

Notice that at this point we have managed to define every integer operator in terms of addition. This is not a coincidence. In a moment I will ask you to write methods for almost all of them using only addition and subtraction. First, I will need to explain recursion, and before that we will briefly review the if, then, else statements (and operator).

If, Then, Else

You will want to research if statements if you have not used them before but they are fairly intuitive. The way an if statement works is that when the computer encounters an if statement it checks the boolean of the if statement. If the boolean is true, it performs the associated code. If statements are pretty intuitively simple.

To write an if statement write:

```
if(boolean){
        doStuff();
}
```

If do stuff is one line, the brackets are unnecessary and the ";" at the end of your code will suffice. This is true for else if and else as well.

To write an else if statement is exactly the same. Use the keyword "else if" not "elif". Else is "else"

Any method which is not void must return something. What it returns must be of the return type that the method was declared with. When we type "return x" the method returns x. As soon as the return code is called, the method call exits. Any code written after a return statement is unreachable and referred to as dead code.

Programmers may use return statements within if statements to "break out" of a method if certain criteria are met. They may also use return statements with an if statement to negate the need for an else statement. In an example above we wrote

```
public static boolean rightArrow(boolean A, boolean B){
        if (A && !B) return false;
    else return true;
}
```

but we could have written

```
public static boolean rightArrow(boolean A, boolean B){
        if (A && !B) return false;
    return true;
}.
```

If we truly want to write short code we could just write

```
public static boolean rightArrow(boolean A, boolean B){
        return !(A && !B);
}
```

and there are even other variations which I challenge you to find.

Recursion

We call a method **recursive** if the method calls itself. Take for instance the following code:[8]

[8] it is considered good practice to capitalize the first letter of words in a method name, excluding the first letter - lowerCamelCase

```
public static void runsForever(int a){
        if (true){
                System.out.println(a);
                runsForever(int a);
        }
}
```

This code is really fucking stupid. When it runs it checks to see if true is true (Java knows "true" and "false" as booleans). Then it calls itself. This will print whatever int is passed to it to the console forever.[9]

[9] Actually it will stop when it runs out of memory and throw a "stack overflow" exception.

Now I am going to walk you through writing recursive addition using subtraction and addition. This code will serve no purpose except to explain recursion better. First, please note that $a + b$ is equivalent to $(a + 1) + (b - 1)$.

Get started by making a new class (it can be in the same project or you can even start a new project, which I would not recommend). You don't even need to make a new class you can type this over your Hello World code (don't. Start over from the beginning and remember to include a "main" method). Do not delete the main method, it is still going to be necessary.

Now let's come up with a name for our method. I'm thinking recursiveAdd will suffice. What arguments will it take? It will need to take two numbers as arguments. Lets make the argument type for both, int (we need to declare the argument type in java, just like we need to declare the return type). Is recursiveAdd a public method? Yes. Why not. Is it a static method? If we want to call it in the body of main (which is a static method) it will need to be static, too.

You cannot make a static reference to a non-static method. I won't explain this yet, for now just make the method static. Feel free to research it on your own though. What will recursiveAdd return? It will need to return a number, let's make this return type int as well.

What do we have so far? "public static int recursiveAdd(int a, int b){}." Now we need to write some code to add them. The most important thing to remember in recursion is that you need a base case to stop your recursive method from running forever like in the really fucking stupid method above. We will recursively add a and b by adding $a + 1$ to $b - 1$. When will we stop? We want to stop when $b = 0$ ("\\" is used for putting comments into code).

```
public static int recursiveAdd(int a, int b){
        if (b == 0)
                return a; \\ on this call a should be equal to a + b
}
```

OK we have written our base case. Now we need to make the code do stuff recursively. As I mentioned we want to add a to b by adding $a+1$ to $b-1$ until $b-1=0$. Lets do just that.

```
public static int recursiveAdd(int a, int b){
        if (b == 0)
                return a; \\ on this call a should be equal to a + b
        return recursiveAdd(a+1, b-1);
}
```

Can you see how this works? If we pass it a = 1 and b = 2 it will do the following

```
1. return recursiveAdd(1+1, 1);
2. return recursiveAdd(1 + 1 + 1, 0);
3. return 1 + 1 + 1; \\ or 3 as it is commonly known.
```

Is there anything wrong with the code? If you said "no," you're wrong. What if we pass $b = -1$?[10] We must deal with this case but how we do it is left up to us. I choose to think that recursiveAdd should only be used on positive integers. However, we will actually solve this case explicitly.

[10] an infinite loop will form, make sure you can see why

```
public static int recursiveAdd(int a, int b){
        if (b == 0)
                return a; \\ on this call a should be equal to a + b
        if (b < 0)
                return recursiveAdd(a - 1, b + 1);
        return recursiveAdd(a+1, b-1);
}
```

Notice that I don't use else if statements. Fuck 'em. While that might be bad practice, I have rarely seen them used in industry. This is not because they are useless, but rather because the sacrifice in speed for using "if" instead of "else if" is compensated for by the decrease in complexity for human beings to read and understand the code. I am also using the fact that return will always break out of our method and any code that happens after return will be forgotten. This means that the only way to get to the second if is not to fall into the first if, which is the same usage as else if.

To verify that this method works run the code as follows

```
public class YourClass {
        public static int recursiveAdd(int a, int b){
        if (b == 0)
```

```
                return a; \\ on this call a should be equal to a + b
        if (b < 0)
                return recursiveAdd(a - 1, b + 1);
        return recursiveAdd(a+1, b-1);
        }
        public static void main(String[] args){
                System.out.println("3 + 8 = " + recursiveAdd(3,8));
                System.out.println("5 - 4 = " + recursiveAdd(5,-4));
        }
}
```

Now, an assignment for you. Write a recursive function for $a * b$.[11] Also, make an exponential recursive function which raises a^b.

[11] Hint: Java will treat your method as if it is of the type that it is returning. That's why we were able to return it in the previous example. If I had wanted to I could have had the method return $1 + recursiveAdd(a, b - 1)$

More Operators!

[12]

[12] In this case we are using = as the assignment operator.

++ (DOUBLE PLUS): for an int a, a++ is equivalent to a += 1 is equivalent to a = a + 1.

– (DOUBLE MINUS): for an int a, a– is a -= 1 is a = a - 1.[13]

[13] These operators are called incremental operators and they are frequently used to keep track of how many times a particular process has run.

?:: this is called a **ternary operator**. For a boolean q and two objects a and b, q?a:b will return a if q is true, and b if q is false.

< and <=: quite literally "less than and less than or equal to."

> and >=: quite literally "greater than and greater than or equal to."

Iterative Solutions

There is a type of recursion called tail end recursion. If a recursive method is tail end recursive, then it can be written iteratively (with a loop).

Read this to understand tail recursion: http://stackoverflow.com/questions/33923/what-is-tail-recursion

I will now teach you how to use for loops in Java. The for (s : set) style works (we can even make our own classes that follow this style) but we will not focus on it. Instead I will show you what is different about for loops in Java. Then you can research **while loops** on your own (a daunting task!), and **do-while loops** on your own. (DO THIS). We are going to learn by example. I'm going to show you some code and we're going to dissect it. As a challenge to yourself you can attempt to implement each of these examples in a while loop.

```
int sum = 0;
public static int sum(int a){
        for(int i = 1; i <= a; i++){
                sum += i;
        }
        return sum;
}
```

The first thing you might ask yourself is: what does this do. It sums all of the integers up to and including a. The next thing you might ask is, why did you write sum out of the **scope** of the sum method? I did this to show you that you are allowed to do this. It makes more sense to put the sum placeholder in the sum method. Why? Even when we do not call the sum method, we have wastefully allocated memory to the sum integer. Further, if we call the sum method a second time within the same application, it will be wrong. This is because it will still retain the sum of the previous call. Do not do it this way. Put int sum = 0; inside of the method. There will be cases in the future when we want to put stuff like sum outside of the method. Another thing we can do is as follows.

```
int sum;
public static int sum(int a){
        sum = 0;
        for(int i = 1; i <= a; i++){
                sum += i;
        }
        return sum;
}
```

This time the code is valid, although the declaration of sum before it is used is still unnecessary and still problematic.

Now on to the for loop. What it do? For loops are written like this for(initializer; boolean; operation). If we wanted we could generalize any for loop to a while loop. While loops are written as while(boolean) and will continually loop until the boolean statement is false. The above for loop is equivalent to the following.

```
int i = 1;
while (i <= a) {
        sum += i++;
}
```

What the for loop is it is declaring the first thing and then running a while loop. After each iteration of the while loop it does whatever is written last. We may also write a for loop without the first and last argument.

```
int i = 1;
for (; i <= a;) {
        sum += i++;
}
```

As an exercise write a method using a for loop which raises a to the exponential power of b.

Arrays

First let's note something about arrays. To get the element of an array, *ary* in location i (where i is an int) we simply want to type ary[i]. By convention arrays start at 0. Thus, ary[0] is the first element in the array called ary.

You already know that arrays are stored in contiguous memory. I don't need to tell you that again. What is interesting about being stored that way is that it is really easy to get from one element to the next. The computer knows where an array is stored and it is simple algebra to hop to an array indice. For that reason it is incredibly fast to get to an element in an array if you know where in the array it occurs.

We have talked about arrays. Now lets use one. Here is a method that sums all of the numbers in an integer array:

```
public int sum(int[] ary){
        int sum = 0;
        for (int i = 0; i < ary.length; i++){
                sum += ary[i];
        }
        return sum;
}
```

The ary.length returns an integer which corresponds to the size of the array. Because arrays begin with 0, ary[ary.length - 1], is the last element in an array, *ary*. This is why we only consider i strictly less than ary.length.

The following are examples of syntax for declaring an array:

```
int[] ary = new int[10]; \\ creates an array of size 10
int[] ary; \\ creates an array which still needs to be initialized
int[] ary = {1,2,3,4,5}; \\ creates the array with elements 1,2,3,4,5
```

NOTE: The next questions almost all involve primes. You may want to make a boolean method called "isPrime"

WHICH RETURNS TRUE IFF AN INTEGER IS PRIME AND RECYCLE IT THROUGHOUT.

1. Now it is your turn. Write a method which prints the first hundred primes. [14]
2. Now write a method which logs the first 10 primes into an array, then sums all of the even indexed primes (you should onsider 2 to be the first prime). This should be 3 + 7 + 13 . . . (ignoring 2, 5, and 11).[15]
3. Now write a method which does the same thing without using an array.[16]
4. Write a method for finding the largest int in an int array.
5. write a method for determining if an array is a palindrome.
6. Euler Problems: 1-3 (for 2, do not use recursion; for 3, you will want to incorporate the use of longs (as opposed to ints) into your solution).
7. **Super Extra Credit:** Euler problem 35.[17]

[14] Hint: you only need to check the numbers up to $p/2$ to determine if p is prime. That is if p is not divisible by an element less than $p/2$, then p is not divisible by an element greater than $p/2$. You will want to use the % operator to determine divisibility

[15] Hint: you may want to initialize an int to use as a pointer to an array index

[16] Hint: you may want to initialize an integer to use as a counter

[17] Hint: This problem is an optimization problem. Run your code on 100 to make sure it works before ever using any larger number. You will want to find a way to decrease the run time in this problem. It took my computer 5 minutes to solve this with my first solution. It takes it 3 seconds now that I have optimized it. I can still optimize it further. Feel free to stop if you manage to make it work for the first 100 integers. Optimization is not something I would recommend focusing on at this point. Remember what / and % do

Doubles, Arrays, Strings, Lists, and Bit Operations

A Brief Tirade on Doubles

I really fucking hate doubles. Doubles just enable us to do some nice neat mathy shit with decimals. There are problems associated with doubles though. Because they are stored discreetly but represent irrational numbers, two doubles might be logically equal (lets say they are really both equal to some x/y) without being equal in value. For this reason we don't use the == operator on doubles. Instead we write something like (x - y < .00000001) so that x and y are equal up to some tolerance for error. Beyond that I really have no desire to talk about the various issues related to doubles. Read about them on your own. If you feel courageous try to understand the concept of machine epsilon.

Learning to Effectively Search the Internet

If you are not familiar with the Math framework, become familiar with it. You use it whenever you use Math.method. One of the nicest things about Java is that Oracle provides you with a great **API** (application programming interface). For your basic data structures you can summon them from the Collections framework (import). But what is this API and what the fuck is an interface? Interfaces belong more to the idea of object oriented programming. In Java you have classes and interfaces but an interface isn't a term for something visual like a UI (user interface), it is actually a set of methods. The iterator interface has three methods.

```
hasNext();
next();
remove();
```

That's all there is to an interface, but I don't expect you to understand the relevance yet. What I do want from you is to learn how to google things better using the Java Oracle documentation. As we start using Strings, StringBuffers, BufferedReaders, and other necessary java classes I want you to be able to find the method you need without relying so much on websites like stackoverflow.

Using the Java Oracle notation is easy; understanding it is difficult. Over time you will fill in the gaps but for now do your best to utilize it. Let's say for instance you have a string, theString. What methods can you perform on theString? If you type theString. (with the period after) eclipse will begin to try to guess what method you'd like to use (ctrl + space also should work). You may see a method called contains() and think "that's perfect." Maybe it is. Maybe it doesn't do what you think it does. How do you find out? Google: "String oracle," and click on a link to oracle.com. There's a common saying in the tech industry: **RTFM** or read the fucking manual.

http://docs.oracle.com/javase/7/docs/api/java/lang/String.html

You can see using this example that oracle first describes the string class. Under the list of constructors are different means of construction for that object: to construct a new String type "new String(argument);"

Depending on the argument you use the string is constructed differently, but the **new** key word is how you declare a new object to be placed in memory. As for the allowed methods you can scroll down further to the list of methods, "Method Summary." Here you will see a table with what the method returns on the left and the method name along with its argument on the right. Beneath each method is a brief description of it. Click on a method to learn more about how that method works.

Terms and Keywords

TERMS:

- Immutable
- Object
- Primitive
- Inherit
- Reference

KEYWORDS:

- new
- null
- switch

Reading Input From the Command Line

Let's build a method which reads input from the command line and then returns it as a string. We do not have to use a prompt, but we will, because otherwise it is not clear to the user when to type into the command line. I call this method Echo.

```
public String echo() throws IOException{
        System.out.println( Speak  into the aether:    );
        BufferedReader br = new BufferedReader(new
                InputStreamReader(System.in));
        return br.readLine();
}
```

Check out that method! Can you google it on your own? What is a BufferedReader and what is an InputStreamReader? What does the br.readline() method do? Why does it throw an IOException and what is an IOException?

I won't answer these questions. This is the sort of HelloWorld of console input. What I will say is that you may want to make a method that more than echos. In general for that situation you'll want to use a switch statement. The switch keyword acts sort of like a bunch of if statements. In fact you don't really need to use a switch statement but it's nice and convenient. The way it works is that switch takes a variable like this (assuming that it's a string)

```
switch (variable) {
        case "1": System.out.println("1 is a stupid choice");
                break;
        case "2": System.out.println("2 is a good choice");
                break;
        default: System.out.println("I only care about 1 and 2");
                break;
}
```

It is not difficult to see what is happening here. The switch statement is checking to see if variable is 1 or 2. If it is neither it lands on default. That's sort of like an else statement. What's interesting is that this works for Strings. If you read up on the switch statement on Oracle you will understand that this isn't a fluke, but you probably won't understand why I think it's worth noting. You see Java passes all objects by memory address (reference). If we have two Strings a and b, and we check a==b, it is only true if a and b are EXACTLY the same object. The strings could be identical and this might return false. This is true of all objects (non-primitives). For this reason when comparing objects we use the .equals(); method. In this case we would check a.equals(b);

Arrays, Again? Also, Strings.

Arrays! We did these in the last lesson. What is an array? An array is literally contiguous memory. When we declare an array we tell the computer to assign some amount of memory to it. When an array stores anything besides primitives (recall what these are from the previous lesson) then it does not explicitly store the object. If we have an array of strings, we don't really have an array of strings. We have an array of pointers to string objects. For this reason arrays of objects which take up a lot of memory, still take up the same amount of memory as arrays of objects which take up a small amount of memory.

Strings are basically arrays of char. You can see this near the very beginning of the Oracle documentation for strings.

Declaring an array:

```
int[] array = new int[x]; where x is the size
int[] array; where array has not been initialized
String[] array = new String[x]; a string array
String[] array = {"hi", "cat"}; explicit initialization of a 2 array of strings
```

What about Strings?! Strings are essentially arrays! (as I have said) A String is an **immutable object**. What does that mean? It means it can not be modified once it has been created. Once we declare an array we can't go back and say, no array, I want you to be bigger than you were before. Similarly once we have a string we can't change it. You may say, "but yes we can. I do it all the time. I just had a string earlier where I did this"

```
String temp = new String();
temp += "y";
```

Why does that work if the string is immutable? Well the truth is, you aren't actually changing the string. A new string is being declared and all of the old string is added to it and then whatever you are adding to the end. Essentially a String is a sort of **wrapper** for a dynamic array. It's not, really, but it sort of is.

A StringBuffer is a truly **dynamic array** class. StringBuffers are like strings except they are mutable. They can be altered without declaring a new StringBuffer. If you would like to learn more about how this works I recommend reading up on it in the Oracle Documentation. The point being that if you have a method where you are continually altering a string, it will likely be faster and more memory efficient to use a StringBuffer instead of a String. At the end you can still cast a StringBuffer to an immutable String.

Strings and StringBuffers have nice features. When reading from the command line, or even files on your computer, if you read in the items as one of these objects you will not have to worry about the size of the input overloading the max size of the object (unless you literally run out of memory) and you will be able to perform methods on the object which will allow you to utilize the information better. Strings have a method which allows them to return a substring, and a method which allows them to return an array. If you use the split() method on a string it will return an array of strings which occur between each argument passed to split(). If the argument of split is "", then you will return an array of individual letters from the string. If you use " " then you will return an array of words. Be wary in the first case as the first "" occurs before anything else in the string and the array returned will have a unique element at the 0 index. Will that element be **null** or an empty string and why?

The null element is actually just a pointer to the location 0 in memory. That's why we generally say "points to null" not "is null" when discussing the null pointer. Actually, I will never call it the null element again.

Matrices: Arrays of Arrays

As I am sure you worked with these in highschool I will be fairly brief. The term matrix, is in my opinion, unnecessary, for the purposes of computer science. A nxm matrix is more easily visualizable than an nxmxl matrix. That is because the latter matrix would need a three-dimensional representation. You can think of an nxm matrix as a table with n rows and m columns. In this regard they are useful for later data structures, but generally speaking an nxm matrix usually is unnecessary over an array of size n*m. However, you may find yourself declaring a matrix which does not have a rectangular analogue. For this reason we shall refer to matrices as **arrays of arrays**.

The declaration is simple. Instead of int[] we will use int[][]. All we are doing is making an array of arrays. The latter box represents the array containing the arrays, the inner box the individual array (I think).... honestly even at this point in my life I will still confuse which box is which. That's why the internet invented stackoverflow.

Whereas in python arrays are dynamic and allow for different objects to be places in them, a Java array only allows the objects you declare the array to be made out of originally and is a fixed size. If we wanted we could bypass both of these constraints in Java, but they requires us to understand how to create our own objects (i.e. OOP). However, if we just want our array to take different kinds of objects it is actually not too complicated.

You see, in object oriented programming there are hierarchies; Things which subclass each other. For example the BufferedReader constructor takes a reader as an argument but we used an Input-StreamReader. There are many kinds of readers and they all inherit from the Reader class. If we wanted to we could make an array of Readers and have within that array all kinds of different readers. Or we could do something more crafty and make an array of Objects.

In java every class inherits from Object (meaning falls lower on the hierarchy than Object). This means that if we make an array of objects we can put anything we want inside of it. What then is the problem? Getting those elements out. That leads to an issue called type casting and we will return to it later.

Dynamic Objects: ArrayLists and Vectors

While it irks me in a serious way to introduce you to a proper data structure before you even know object oriented programming and can implement it on your own, the Collections framework provides you with better, faster versions of the data structures you will later learn to implement.

The difference between a Vector and an ArrayList is not one that I can explain well until we understand multi-threading a little bit. Vectors are considered **thread safe** and are essentially the same thing as ArrayLists so for now we will choose only to think and talk about the ArrayList.

What is an ArrayList? Look it up on the Oracle notation but let me tell you a little first. An ArrayList is a dynamic array. Beneath the object there exists an array which keeps track of its own capacity. When you try to add an element to the ArrayList and there isn't enough space left to do so, the ArrayList declares a new array (likely 2x the size of the old one), copies over every element from the previous array one-by-one and then adds your new element. If space exists it just adds the element to the end of the array.

What is nice about ArrayLists is that they implement all of the methods that a list implements. You can insert an element, add an element, remove an element, whatever. The problem is that an ArrayList, at its core, is still an array. Inserting an element requires us to one-by-one move every element over to make space (and even possibly resize). Removing an element is similar. In the worst case we may need to move every element over and resize. That means that the time it takes to perform this could be said to be proportional to the size of the array. However, adding an element to the end of an ArrayList still takes negligible time as even though the array list may need a resize (which would take time proportional to the size of the array), this happens so infrequently that we do not generally consider it as particularly relevant.

Using an ArrayList is pretty simple but understanding how to declare it will help us understand how to declare objects in Java which are non-primitives and not so basic as String objects. We will need to import ArrayLists in order to use them. To declare an array list of Strings we do the following.

```
ArrayList<String> the_list = new ArrayList<String>();
```

What is happening with those brackets? The ArrayList takes a separate argument for the type of object it will contain. Almost all objects which require a type of object to be specified are written this way. We can pass any non-primitive into these brackets and the ArrayList will be of that type. If we wanted we could make an ArrayList of Objects and have something more similar to a python array. However, we cannot instantiate an ArrayList<int> because int is primitive. How do we work around this?

For every primitive there exists a wrapper class. In the case of int the wrapper class is Integer. If we make an ArrayList<Integer> we can add ints to it and Java will do the work of actually turning them into Integers. For shorts the wrapper class is Short. For longs the wrapper class is Long. These wrapper classes simply contain the primitive classes and some methods that are considered beneficial. I can imagine the idea of a class containing things is a bit difficult to imagine. This is sort of the crux of object oriented programming. We create our own objects with their own methods and fields.

The Cosmic Object

Every non primitive inherits from the cosmic object. I wrote this and then talked about the cosmic object above but it's worth noting again. By default every object inherits from the cosmic object. Hence the name. We can declare ArrayList<Object> and add any element we want to our array list. Unfortunately, it will still be difficult to retrieve an item from our ArrayList. We can't just say

```
ArrayList<Object> theList = new ArrayList<Object>();
theList.methodWhichAddsElementsOfAllTypesToOurList();
for (int i = 0; i < theList.size(); ++i){
        String a = theList.get(i);
}.
```

What if the ith element of theList isn't a String? That would be bad. We will return to this issue later. For now if you do decide to do things like this it would be best to know exactly what you are returning. For instance, System.out.println(theList.get(i)); will always do something. What will it do?

Lists

Let's look ahead now. A list is a data structure. An abstract list can have a few meanings but here we are referring to it abstractly as a data structure, so we are actually just talking about theory. How we implement a list is irrelevant from what we mean by **list**. In general we mean some kind of data structure which holds data in an ordered fashion.

In Java there is a list **interface**. Feel free to check out the Oracle reading on it but interfaces still are pretty far ahead of us at this point. In some regard the methods that the list interface contains are what Oracle feels describe the functionality, abstractly, of a list data structure.

Here we will discuss the concept of a **linked list**. Now we are really getting into the meat of things. Again, we are just talking about list data structures. I could introduce linked lists and **doubly linked lists** the same way I did arraylists but instead I will just tell you how they are implemented and let you try to figure out when to use them. Later, we will talk much more about them.

A linked list is a list data structure with all of those nice list methods (use Oracle) but underneath all of this a linked list is really a bunch of list nodes. These list nodes contain a generic data type (the objects they store) and a pointer to another linked list node. In a singly linked list each node has a pointer to the next node and that is it. What's good about this? Insertion is really simple. We don't actually need to move things over we just put a node in the right spot. What's slow? Well the way the computer passes linked lists is by a reference to the head of the linked list. In other words if you want to find an element at location i, unlike in array lists, it requires you to iterate one-by-one through the list until you get there. Adding elements to the end of the list is similar. We must start at the beginning, work to the end, and then append the last element. However, appending elements to the front of a linked list is incredibly fast. Operation on the front of a singly linked list are really fast in general.

```
LinkedList<Object> the_list = new LinkedList<Object >();
```

Doubly linked lists are exactly the same as linked lists except each list node has a pointer to next element and a pointer to previous element. The added functionality is that we can traverse backwards and forwards. The cost is that each list node now has more memory allocation. Doubly linked lists are less frequently used than you would think. Most of the problems solved by adding a backwards link can be solved using a singly linked list with a more intelligent algorithm.

Some More Bitwise Stuff

The computer stores data in actual binary. All bit operations work in boolean logic. They treat a 1 as true and a 0 as false. In that regard 0001 'and' (&) 0000 is 0000. That is to say 0 & 0 is 0, 0 & 1 is 0, and 1 & 1 is 1. This may seem complicated. We are returning the truthiness of the statement "true and true," which is true for "1 and 1." Here are the basic bitwise operators.

(http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html)

Operators

- «: shifts everything to the left (puts a 0 at the end). [18]

  Ex: a « b shifts the bits in a over, b times. Thus, 2 « 1 is 4 (represents 0010 « 1 = 0100).

- »: shifts to the right, may put a 0 or a 1 in the leftmost position depending on the sign of the int.

- »>: shifts to the right, will always put a 0 in the leftmost position. Where does Java store the sign of an integer in the binary? Which bit contains that information?

- &: Similar to the 'and' that you are familiar with, but one ampersand instead of two. If we 'and' two ints, wherever they are both 1, the result will be 1 and everywhere else the result will be 0.

- ˆ: Generally called the xor operator this is the 'exclusive or' operator. When we say (a or b) sometimes it is unclear what we mean.[19] In the case of exclusive or (or xor) we only allow for (t^f) and (f^t) to return a true value. In this regard when we xor two ints, wherever they both differ the result will be 1 and wherever they are the same the result will be 0.

  Ex: 1010^0101 = 1111 and for an bit or int a, a^a = 0.

- |: The regular 'or' operator we are used to. If we 'or' two ints, everywhere where either one of them has a 1, the result will have a 1. The result will be 0 everywhere where both ints had 0s.[20]

- : Called the unary bitwise complement operator. We will call it the 'not' operator. When we not an int we just reverse every bit. The reason we call this a unary operator is because it acts only on one int. In every example above the operators required two arguments. They were binary operators, or binary bitwise operators (not binary binary operators).

  Ex: 1010 = 0101 and a^a = 0, always.

[18] In binary this is the same as multiplying by 2 but because of the way ints are represented in binary the results may not be the same. The leftmost binary number can store the sign, depending on the data type.

[19] I have told you that in Java if turns out to be true, Java doesn't even bother checking b. This is the non-exclusive or version.

[20] The choice of representation is arbitrary. If we chose to represent true as 0 and false as 1, but demanded the | operator act in the same manner, the | operator would just be the 'and' operator. What I'm saying is, & is to 1 as | is to 0.

Ex: 1010 | 0100 = 1110. So as you can see the result is only 0 wherever both a and b are 0.

It is important to be able to interpret binary numbers. Numbers written in any base are just a variation on modulos in some regard. When we work in base n, a number a % n is always the last digit representation of that number.[21] To retrieve a number from binary and turn it into base 10 we can describe an algorithm. We will choose to write our binary numbers the same way we write our base ten numbers. The leftmost digit will be of the highest order and the rightmost digit will be of the lowest order. When we write a number base n we write it as a series of symbols or digits $A = a_m - 1, \cdots, a_1, a_0$ where the symbols $0 \leq a_i < n$ for any $i < m$ and the series A is m in length, the series represents a number given by

$$(a_{m-1} * n^{m-1}) + \cdots + (a_1 * n_1^1) + (a_0 * n_0^0)$$

.

[21] I say digit, but digit is the wrong word. The word digit refers to base 10. If we are working base 2 the last digit is really called the last bit. However, we will use the word digit to refer to a symbol in a system which refers to a number strictly less than the base it is being represented in. This is not really math; it's meta math. We are talking about representations of numbers, not the numbers themselves.

In this regard, the last digit is always the one's place. In binary we can take a number $b_7b_6b_5b_4b_3b_2b_1b_0$ in base two and convert it to base ten by multiplying each digit by $2\hat{i}$, where i is the index in the binary sequence, and adding them.

$$10101010 = 1 * 27 + 1 * 25 + 1 * 23 + 1 * 21 = 128 + 32 + 8 + 2 = 170$$

[22]

[22] This is a byte (8 bits) and as you can see it isn't capable of storing numbers which are all that large.

Note that $2k = 2 * 2k - 1$ by simple math. Because Java stores the sign of a primitive in one of the bits[23] the maximum value of a signed primitive is strictly less than 2k where k is the number of bits in the primitive. So what is the maximum value of a binary sequence of length $2k - 1$? Well it is 1 less than the value of 2k. If you can't see this just imagine we have a sequence of all 1s with a leading 0. If we add 1, the sequence is just a leading 1 followed by all 0s. $0111 + 0001 = 1000$, is the same as $0999 + 0001 = 1000$ in base 10 (different numbers, same concept). For this reason the maximum value of a primitive with k bits is $2k - 1$ but what is the smallest value? The smallest value is expressed as all 1s, but it would be silly if it were the same size. When we considered all the positive values we included 0, but negative 0 is still 0 and there is no reason for 10000000 to be equal to 00000000, so the smallest value is just -2k for a primitive with k bits.

[23] you should know which

Common Bit Functions

- Get Bit
- Set Bit
- Clear Bit
- Update Bit

Write these methods on your own, expressing a number in any base you'd like.

```
public ArrayList<Byte> toBaseN(int number, int base){
        ArrayList<Byte> the_number = new ArrayList<Byte>();
        int temp = number;
        while (temp > 0){
                the_number.add((byte) (temp % base)); //See explanation.
                temp /= base;
        }
        return the_number;
}
```

This method will spit out an array list of bytes, thus our base should never exceed the maximum size of a byte + 1 (128). The Byte is the wrapper class for byte, making it non-primitive. We cannot declare an ArrayList of primitives, as always.

The line I marked "see explanation" has a weird argument in the add() method. In the argument it would be sufficient to just write temp % base if the ArrayList were of Integers, as temp % base returns an int. However, the add method for an ArrayList of Bytes needs to take a Byte or byte argument. In order to do this we are typecasting the int returned by temp % base to a Byte. This is the syntax for type casting, a concept which we will discuss later. What will happen if our base is something larger than 128, for instance 1000? We can avoid this case by adding code to the beginning of our method which stops it from attempting to run in that case.

What we have done is we have said "treat this int as a byte." This is not a problem because a % b returns a positive and because what is returned is always less than 128, meaning the leading bit of the newly casted byte will always be 0 (so we don't have to worry about negatives). This does not work for negative numbers as the while loop will never run. I never like to give you fully functional code. Your job is to see what's wrong with it and think about fixing it.

You may also feel that this particular method of storing a number is not to your liking. The representations of digits larger than 9 will still occur in base 10, thus if our base is 11, we may find some of our digits are represented as 10. If our base is 128, some of our digits (some of the elements in the array list) may be as great as 127. We also have our lowest order digit occurring at index 0, which is a choice that may or may not be a good one depending on the intended usage.

PRE-EXERCISE QUESTION: What is the worst case runtime of this method? That is, where n is the size of the number, and m is the size of the base what is $O(n, m)$[24]? This is a somewhat difficult question. Ask yourself how many times the loop runs when m is 2, now 3, now generalize to m. In multi-variable math it is often easier to see what the function looks like in one variable for specific cases of the other.

[24] that's right, the function will be in both n and m

For instance $f(x, y) = x^2 + y^2$ is sort of like a cone. We know that $x^2 + y^2 = r^2$ is the equation of a circle with radius r, but when we consider $x^2 + y^2 = z$, and allow z to be variable, the problem becomes more difficult. All we need to do is see that for any z, we have a circle of radius $sqrt(z)$, so as z increases we have larger and larger circles. The radius increases as $sqrt(z)$ so the function isn't really a cone. It's more like what would happen if we took $z = y^2$ (or $y = sqrt(z)$) and rotated it around the z axis to make a solid shape. Recall that $y = x^2$ is the formula for a parabola. This is called a hyperbolic paraboloid and we have only considered its behavior in the positive z direction.

http://ltcconline.net/greenl/courses/117/DerivNvar/surfac1.jpg

Exercises

1. What do I mean when I say Java passes objects by reference?

2. Make a program using a switch statement somehow. Impress me.

3. Let's say I wanted to insert a number into an abstract list of numbers. What data type would I use to do this? Array, ArrayList, LinkedList, or String? Why?

4. Write a program which takes a number and turns it into an ArrayList of digits.

   Example: 13425 should give the ArrayList 5,2,4,3,1 or 1,3,4,2,5 depending on your preferences.

5. What benefit could be obtained by expressing a number as an ArrayList instead of as an Array, String, or primitive?

6. I assume you used an ArrayList<Integer> (Integer is the wrapper class for int). Are there any other data types besides Integer which would work? Why would(/wouldn't) you use that data type instead?

7. Project Euler Problem 16. Try to do this without using Strings.

8. Project Euler Problem 22. To sort a list you can use the Collections.sort(list) function.

BONUS: What is the run time of the following algorithms?

9. What is the worst case run time of inserting an int into an array? Describe the worst case situation.

10. What is the worst case run time of inserting into a string? String buffer?

11. Describe the worst case situation for inserting an int into a singly linked list. Describe the best case.

12. I've given you an array of ints. I ask you to sort them. Write a pseudocode algorithm to sort the array. Describe the worst case scenario. Describe the best case scenario.

Object Oriented Programming

Terms

- **inherit** - A subclass inherits methods and fields from a superclass.
- **subclass** - A variation on a class which shares all of the same fields and methods but also has its own fields and methods.
- **superclass** - A superclass is just the opposite of a subclass.
- **constructor** - A method which initializes an instance of a class.
- **override** - A subclass or a class which implements an interface (or both) can override methods.[25]
- **overload** - Two methods of the same name which differ in type or argument. Java knows based on the usage which one you are referring to, or else you fucked up.
- **garbage collection** - Research this.
- **interface** - A set of methods and variables which you can declare in the interface. Methods are declared but not implemented in the interface.
- **abstract class** - A class which can not be instantiated outright. It can be subclassed and generally has its own abstract methods which can be overridden by subclasses. Similar to an interface, but a class.
- **type casting** - Taking a superclass and telling Java to treat it as one of its subclasses. It is important to typecast to the correct class.

[25] The method signature must be the same (return type, keywords, arguments). By changing the method body, we can change what the method does on an instance of the class.

Keywords

- **abstract** - Can be called and written by subclasses.
- **public** - Can be called and (for variables) modified anywhere.

- **private** - Can be accessed by methods of the same class, inner-class methods, and inner-classes of the outer-class.
- **final** - of a variable: cannot be changed. Of a reference: cannot point elsewhere. Of a method: cannot be overridden. Of a class: cannot be subclassed.
- **static** - Can only be called by the same class. Ubiquitous throughout the class.
- **protected** - Only classes in the same package can use it.
- **try / catch** - Tries to execute code and dictates behavior upon encountering specific errors.
- **finally** - Always executes after the try block exits.
- **finalize** - E.G. "protected void finalize() throws Throwable." **Garbage collection** calls finalize() right before deleting an object. It can be **overridden** to add exit behavior for the object.
- **throw(s)** - For error handling. Research this.

What is an Object?

This is sort of a tough question. This entire time we have been creating objects and you didn't even care. Basically, an **object** is an instance of a class. A string is an immutable object. We have discussed this, but there are other kinds of objects.

Let's say we want to make our own car object in Java. Better yet, let's say we want to make a program for a dealership so that they can keep track of their inventory and all of the other shit that they have to keep track of when running. How would we do that? Well we'd make a new .java file and name it Dealership.java or something. In there we would declare it as a Public Class Dealership, just like eclipse does for us automatically. As an example I wrote a dealership class.

```
public class Dealership {

  public int register = 0;
  public Car[] inventory;
  public Employee[] salespeople;
  public Person[] occupants;

  // class
  public class Car {
    public String make;
    public String model;
    public int value;
    public boolean sold = false;
  }

  // class
    public class Person {
    public int wallet;
    public String name;
  }

  // class
  public class Employee extends Person {
    public int sales_this_year;
    public int charisma;
  }

  // Dealership constructor
  public Dealership(Car[] inv, Employee[] workers){
        inventory = inv;
        salespeople = workers;
        occupants = workers;
  }

  // methods
  public void addPerson(Person the_person){
    int length = occupants.length;
      Person[] temp = new Person[length + 1];
      for (int i = 0; i < length; ++i){
      temp[i] = occupants[i];
    }
    temp[temp.length] = the_person;
    occupants = temp;
  }

  public Car sellCar(Car the_car, Employee seller){
    for (int i = 0; i < inventory.length; ++i){
      if (inventory[i].equals(the_car) && !inventory[i].sold){
        inventory[i].sold = true;
        register += inventory[i].value - 5;
        seller.sales_this_year++;
        seller.wallet += 5;
        return inventory[i];
      }
```

```
      }
      return null;
    }

    public void hire(Employee candidate){
      String the_string = "";
      if (candidate.charisma < 4) {
        the_string = "Sorry "
          + candidate.name
          + ", we just arent hiring.";
      }
      else {
        the_string = candidate.name
          + ", you're hired!";
        Employee[] temp = new Employee[salespeople.length + 1];
        for (int i = 0; i < salespeople.length; ++i){
          temp[i] = salespeople[i];
        }
        temp[temp.length] = candidate;
        salespeople = candidate;
      }
      System.out.println(the_string);
    }
}
```

The first thing we need to know about classes is that they contain fields. In my version of a dealership I have included the following fields.

- register
- inventory
- salespeople
- occupants

Just like how the line of code "array.length" returns an int value of the array length, if we instantiate a Dealership called dealer, we can find the register value by finding "dealer.register". Unlike an array, because we have declared all of these fields as public, we can also change the values.

In essence what makes up a class is a set of fields and a set of methods. The methods declared within a class are generally meant to act on the class. It is atypical to leave the fields of a class as public. We would generally make the fields private, meaning that they cannot be altered by other classes.[26]

[26] Imagine if you could change the length field of an array. That would be bad, wouldn't it? The array would think whatever is next to it in memory is part of the array if we made the length longer. If we made it shorter we would merely truncate the array.

Classes also must have a constructor method. Java recognizes the constructor method as the method of syntax:

```
keywords classname(arguments){}
```

The constructor method is how the class is instantiated. When you call Dealership dealer = new Dealership(arguments), the constructor method is called and determines the initialization of the class instance. In this case you can see that the dealership constructor takes an inventory and list of employees. It then initializes the dealership to have the inventory passed to the constructor, the employees passed to the constructor, and occupancy of the employees.

This makes sense, we want to consider the employees as occupants and any time we change the employees we want to change the occupants. Within the Dealership class are some nested classes. They act the same as other classes but do not call for their own .java files. None of them even have their own methods.

Some of these classes extend other classes in the file. What does extends mean? It means that an employee is just a more specific kind of person. Employee has the same fields as Person and then a couple more. In this regard we can have an array of Occupants, all of which are people and some of which are Employees. A danger arises however when we try to retrieve elements from that list. We do not know if they are employees or just regular people. Read up on typecasting. I will not describe it here but it is vital.

Beyond that you can see that I have included some methods. These are methods like .size() for a string. When you declare a Dealership you can use these methods on it. Static methods act regardless of instance of a class. Non-static methods are specific to a class. If you declare a method as static, make sure it doesn't modify the fields of your class unless they too are static fields[27]. A static field is the same in any instance of the class.

[27] If they do, it is still potentially a dangerous implementation

Interfaces

In Java we don't have multiple inheritance. This means that no class can extend more than one other class. However, a class may implement from as many interfaces as we would like. Just like classes we can make our data types of type interface. For instance, look up the list interface. If we wanted to we could make the argument of one of our methods a list. Any class that implements the list interface can be passed as an argument to that method. This can be complicated. We may try to do things to the list which are disallowed. It is always important to keep in mind what kind of object you're passing. Sometimes we as programmers choose to program fool-proof methods.

Sometimes we take the knowledge of what is being passed and allow for errors to occur that we are certain can never occur. If you are passing that method a combination of array lists and linked lists then you know what the method can handle, but make sure that it will never be used for other classes which implement the list interface.

Let's Build a Cyclic Array

As an exercise let's make a cyclic array class. This array class will be an array but for a size n cyclic array we will consider the the i^{th} index $(i < n)$ to be the $(i + n)^{th}$ index as well[28]. The array is a cycle. We will want one more quality, though, for our Cyclic Array. We want it to work with the command syntax "for (int a : cyclicarray)", too.

[28] That is to say, the 0^{th} index in the array occurs after the $(n-1)^{th}$ index

In order to do this we need to make our CyclicArray class implement Iterable. Iterable is an interface with one method, the iterator method. The iterator method returns an iterator, which is another interface. In this case we will need to make our own iterator, too. I understand that this is complicated so we will take it in pieces. First we will make a cyclic array class and then we will make it iterable[29].

[29] You may be curious how to make our cyclic array class take objects other than ints. We will get to this using a different example, but for now we will only make an int based cyclic array for simplicity.

```
public class CyclicArray{
  private int[] ARRAY;
  private int HEAD = 0;

  \\Constructor
  public CyclicArray(int size){
    \\Sets ARRAY to new array
    ARRAY = new int[size];
  }

    private int convert(int index){
    if (index < 0){
      index += ARRAY.length;
    }
    return (HEAD + index) % ARRAY.length;
  }

  public void rotate (int shift_right){
    head = convert(shift_right);
  }

  public int get(int i){
    if (i < 0 || i >= ARRAY.length){
      throw new java.lang.IndexOutOfBoundsException(   Index
      out of bounds.    );
    }
    return ARRAY[convert(i)];
  }

  public void set (int index, int value){
    ARRAY[convert(index)] = value;
  }
}
```

At this point we have a functional CyclicArray class. This class does not behave like a list, but rather like an array. We can put elements in wherever we want (for any int index) but they will appear somewhere in the array pertaining to its size. The convert method allows us to convert an index into its proper location in cyclic array as I described before.

The rotate method allows us to change the first element of the array while maintaining a cyclic array. We do not allow for someone to call get(i) on an index i which is larger than the size of the array. This is unnecessary but in my opinion a logical choice. While we could easily translate any input into a proper element in the array, we choose to treat the get method as a method for retrieving an element in a fixed size array, because we are indeed working with a fixed size array.

In general, for any private field in a class we tend to want getter and setter methods. We could easily give the user the ability to modify and check our fields by making them public, but instead we have chosen to make them private and control access to them with a set of chosen public methods. As the user should never need to call convert themselves, we have made convert private. I can't really anticipate any problems with making convert public but I can't see a benefit. Like the standard array, our constructor asks for an array size and nothing else. While you may be thinking, "typically I also have to give the type of array," we have simply chosen to limit our cases to ints in order to focus on interfaces and classes. To make classes which are less restricted we need to learn about generics more and talk about type casting[30].

[30] This can be an extremely arduous process so I ask that you do NOT attempt to make the cyclic array class into a generic class yet. I DO expect you to do this at the end of the section on generics. Discussion of enums and error handling may prove helpful in that matter, as well.

We have succeeded in making a functional cyclic array class but we have not made it work for "for (int a : cyclicarray)" and this is where we learn about interfaces. Just like magic once we implement Iterable properly, the class will work for this syntax. Pretty cool, huh? Let's do it. For the sake of brevity I won't be typing out all of the methods we just typed, again. You can just assume they're there if they're unmentioned.

```
public class CyclicArray implements Iterable<Integer>{
  private int[] ARRAY;
  private int HEAD = 0;

  \\Constructor
  public CyclicArray(int size){
    \\Sets ARRAY to new array
    ARRAY = new int[size];
  }

  private int convert(int index){
    if (index < 0){
    index += ARRAY.length;
  }
    return (HEAD + index) % ARRAY.length;
  }

  \\ More methods.
  public Iterator<Integer> iterator(){
    \\this keyword calls this instance of the class
    return new CyclicArrayIterator(this);
  }

  private class CyclicArrayIterator implements Iterator<Integer>{
    private int _current = -1;
    private int[] _items;

    public CyclicArrayIterator(CyclicArray array){
      _items = array.ARRAY;
    }

    public boolean hasNext(){
      return _current < ARRAY.length - 1;
    }

    public Integer next(){
      _current++;
      int item = _items[convert(_current)];
      return item;
    }

    public void remove(){
      String errorMessage = "The remove function is not supported";
      throw new UnsupportedOperationException(errorMessage);
```

```
        }
    }
}
```

When we implement an interface we must use all of the methods that come with it. The Iterator method demands that we have a method called iterator which returns an Iterator. The interface has a type, and so does Iterator. This is what the iterator iterates over, not the data structure it is iterating through. We cannot have primitives as a type in these cases so we use the Integer wrapper class. While we are essentially done by simply having the iterator method, it is insufficient to do nothing after that. The iterator method must return an object which implements the Iterator interface. The iterator interface includes three methods. The methods are: hasNext, next, and remove[31].

[31] You can probably guess why this is important when trying to make the cyclic array class function for "for (int a : cyclicarray)". It will start somewhere and then call next as long as hasNext returns true. Luckily we do not need to make a remove method to do this.

The next method must have a return type which agrees with the type of the Iterator interface. We use current as a place holder to keep track of how many elements we have cycled through. It is apparently common to use leading-underscores for fields which occur in nested classes. You may test this class. As long as it's in the same package as another .java file, you can even call it as a data structure. It is particularly un-useful but you may find a use for it one day. Making this class function with generics is no small task. You may want to try to find a way to make it so that calls to "for (int a : cyclicarray)" start from the head. To do this you would just need to modify the constructor method so that _items is an array with a different ordering of the elements. There may be other ways, too.

Enums

An **enum** is sort of like a class. Basically it's a way to set values to keywords, or as my friend Zack put it, "more like keywords to values." Enums can have a lot of different uses but here's my favorite. Lets say we have a class Person and we want Nerd, Jock, and Goth to all be subclasses of Person. Person is thus the superclass of Nerd, Jock, and Goth. Now imagine we want to make a School class. *Not that kind of school class.* In that school class we'll want to have a list of students as a field. It makes the most sense to make that a List of People. Now imagine we want to take a person from the list and ask whether they're a Nerd, Jock, or Goth. While each element contains the fields which separate the three subclasses, Java has no way of knowing which is what. Type casting allows us to cast a Person to the right subclass but we need to know what the right subclass is. An easy solution is to create an enum[32].

[32] Or we can call .getClass() on the object. Enums are probably the least applicable or useful means of type casting but I prefer to use it as an example because Enums are kind of boring.

```
public enum SocialClique{NERD, JOCK, GOTH}
```

Now in the Person class we can include a field of type Social-Clique and use its value to type cast our person to the right subclass. This can be accomplished using a switch statement[33].

[33] You may want to read up on enums on oracle, as well as switch/case and type casting if you haven't yet.

Error Handling

What's the best way to error handle? I don't know. That's a design decision. You can make your own exceptions. You can see from above how to throw an exception. Sometimes you throw a new exception. Sometimes you need to declare what type of exception a method throws. Lets say we have a method Q which may or may not throw an exception. If we call it within another method P we have two choices. We can surround our call to method Q (within method P) in a try/catch statement, or we can say

```
keywords methodP(\\stuff) throws BlahBlahException {}
```

and leave the try/catch block to the very end. This is probably the best choice. It is good practice to include a stack trace. This lets the user know where the exception was thrown. I could drone on about error handling, or you could read up on it on your own. As it is not of particular importance to learning conceptual programming, I will leave it to you to follow up[34].

[34] I do want to tell you about try/catch blocks. They aren't really try/catch blocks, they're try/catch/finally blocks. When we use a try block the code attempts what is in the try block and catches errors in the catch block. However, the finally block always runs immediately as the try block exits. The importance of a finally block will remain unclear until you encounter things like threading. For now lets say you want to read a file in a try/catch block. Even if an error is thrown, you may still want to do stuff to the file. You may want to guarantee that the file reader closes. To do this you would write .close() in the finally block.

Generics

To quote myself. "Fuck generics are complicated and easy all in one." You may be better off just reading the oracle documentation on generics to really thoroughly understand them.

For a brief example lets make a weird data structure. Lets make a data structure which is minimally a list. What do we need to make a really minimalist list? A list is just a sequence of ordered elements. Because we want to talk about generics we won't even say what type of element, we'll just say elements. What do we want? We want to make a data structure which contains an element and some kind of sequencing mechanism. You may think a good way to do this is to simply make a data structure of elements and indices. We could potentially do this if we had some other data structure keeping track of the indices, but this is not minimal enough. To maintain sequencing we'll just make each data structure have two fields. It will contain the element we are talking about and a reference to another data structure of the same type[35].

[35] Recall that this is called a linked list.

```
public class ListNode<T>{
  public T data;
  public ListNode<T> next = null;

  public ListNode(T datum){
    data = datum;
  }

  public void add(T element){
    ListNode<T> head = this;
    while (head.next != null){
      head = head.next;
    }
    head.next = new ListNode<T>(element);
  }

  public void print(){
    ListNode<T> head = this;
    while (head.next != null){
      System.out.println(head.data);
      head = head.next;
    }
    System.out.println(head.data);
  }
}
```

We parameterized the generic at the very beginning as T and then just passed T throughout as our data type. This is sort of the essence of how generics work. You can try this class out on strings and ints. Anything that system.out.print will print, the print method of this class will work on. We can be more specific about our generics and that's for you to read up on. For instance we can say that our generic T must extend some other class or be a superclass of something. This version of a LinkedList is a terrible way of writing it, but it serves as an example.

To test your knowledge you may choose to modify our cyclic array class to allow for an array of any type.

Making Our Own Objects

Sometimes it is helpful to make your own objects just to help you with all of your work. For instance we could have a class called Primes with a static method called isPrime. Because isPrime is static (always the same) we don't need to instantiate a Primes object to call it. As long as isPrime is a static method within Primes we can use

Primes.isPrime(potentialPrime).

Make a class called Duration which is capable of checking how long a block of code within it lasts. Ideally Duration.start(); should be called before your code, and Duration.end(); after it. Duration.end(); should spit out the time difference from the start call to the end call. As you can see you will want to make start and end static methods. How will you store the time variables in the class? Will you make them static or not static? Will you make them final or not final? Public or private? You can use System.nanoTime(); which returns a long representation of the time, but try to clean it up and make it as human readable as possible.

Making Our Own Interfaces

Sometimes we want to make our own interfaces. This is complicated and the reasons vary. It is best to look at this through an understanding of **Object Oriented Design** (OOD) and design patterns.

Design Patterns[36]

[36] Read up on these at your leisure, it should be difficult and take time to convince yourself of what they are doing.

- Singleton
- Factory
- Decorator
- Iterator

Exercises

1. Make your own version of array list and call it MyArrayList. It should take a generic argument for the type of data being stored in the array list. The array list will be comprised of at least two fields, an array (initialize to whatever size you want), and an int pointer to either the last element in the array, or the empty spot following it. Make an add method. This method should resize the array to double its current size whenever the array is roughly 75% full. Make a get method which returns an element at the i^{th} index of the array. Make an insert method which inserts an element at the i^{th} index in the array and moves every element after the i^{th} index over one index. Make a set method which changes the value of an element at the i^{th} index of the array.

2. Now make MyArrayList work for "for (<T> a : arraylist)[37]."

[37] You may find it easier to create a private "refactor" method which resizes the array and moves all of the old elements into the new array. This problem should take you a very long time.

3. Using your favorite search engine, how would you make the Collections.sort() method work on an instance of your MyArrayList class? You don't need to do this. Just explain how if it is possible.

4. Project Euler Problem 40

End Note: this is typically the end point for an introductory course in Java. The reason we have been able to move so quickly is because we have focused heavily on teaching through immersion. What we have covered so far is by no means a substitute for a proper Java class in an academic environment, but we are focusing on big concepts rather than precision.