CSC 325 Adv Data Structures

Lecture 1

Java Re-Introduction

I'm sure by now you are all experts in the Java language! But… just in case you still haven't conquered every aspect of it, allow me to go over some Java reminders as well as some Java gems to help you in your quest to become fluent in Java.

Value Types vs. Reference Types

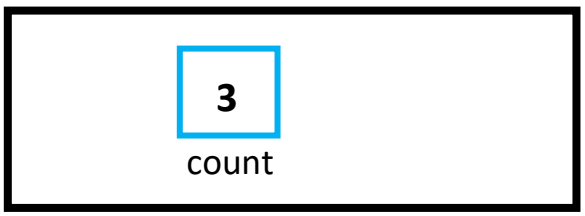
For most of you, this should be a reminder. Java has normal primitive types which just hold values, and reference types which hold memory addresses to values. Here is a nice chart showing some value types and their corresponding reference types:

Value Type	Corresponding Reference Type
int	Integer
float	Float
double	Double
char	Character
boolean	Boolean

The data type on the left just holds a simple value. For example, if we had the following:

```
int count = 3;
```

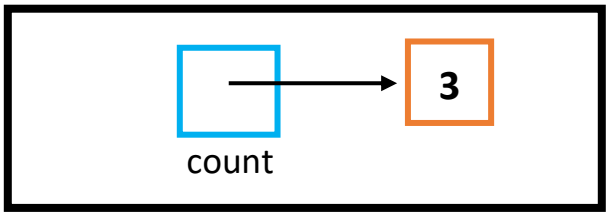
in memory it would look like the following:



As you can see, the value held in count is the value itself. No pointers or references. This is different from the following:

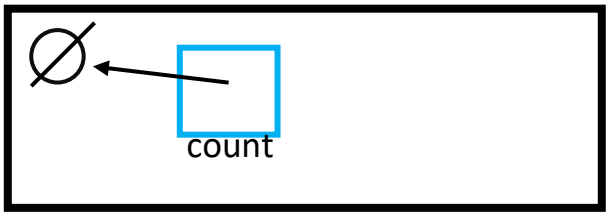
```
Integer count = 3;
```

The line of code above creates something like the following:



As you can see, a reference type "refers" to a value in another part of memory. It holds the memory address (much like the link in a linked list!) of where that value is stored. For this reason, a reference type can be equal to null:

```
Integer count = null;
```



Now the variable count points to null (i.e. nothing). Technically it points to memory location 0x0000 which is memory address zero and is reserved for symbolizing a null value. So, since we can point to memory, we can point to memory address zero, which is what null actually is.

Now, since null is actually a memory location, can we do the following?

```
int count = null;
```

Well, not really. If count is a value type, it can't point to anything. It is only capable of holding a value (an int in this case). And, since zero is a valid int, if we try to make count equal to zero, it won't be null, it will be the quantity zero!

What this means is that, if a variable is a value type, it can NEVER be null! Only reference types can be null. But, you may say, in Python this was possible. Consider the following code:

```
count = 3
count = None
```

In Python, the concept of "null" is done using the keyword "None" (without the quotes). The above code makes it seem as though we are setting count (which is an int) to None (which is Python's version of null). Actually, we aren't! Try this:

```
count = 3
print(type(count))   # shows <class 'int'> on the screen
count = None
print(type(count))   # shows <class 'NoneType'> on the screen
```

As you can see from the output, Python changes the variable's type from 'int' to 'NoneType', which is a special reference type that is allowed to be null.

All classes are reference types. This means that if you create a class called Marker, for example, and you declare a variable such as:

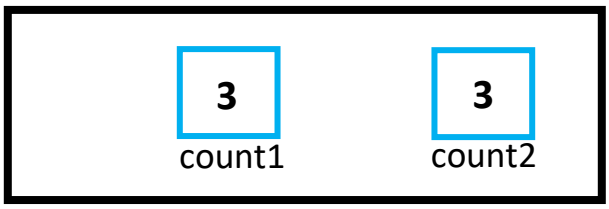
```
Marker m = new Marker();
```

The variable **m** would be a reference type since its data type is a class. The data type String is a class and therefore String variables are reference type variables.

Since reference types are classes and classes all inherit from the Object class, this allows reference types to be used in generics, whereas value types cannot be used (at least not in Java).

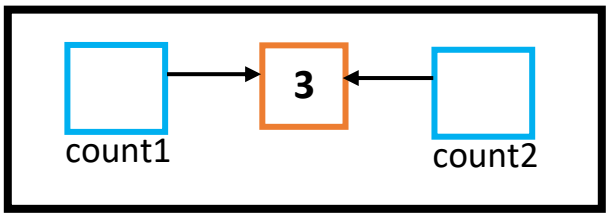
Finally, when you try to copy the value of a variable to another variable using assignment, things are different between value and reference types. Here is what I mean:

```
int count1 = 3;
int count2 = count1;
```



Since value types hold the value itself, that value is copied over. However, with reference types, this is not the case:

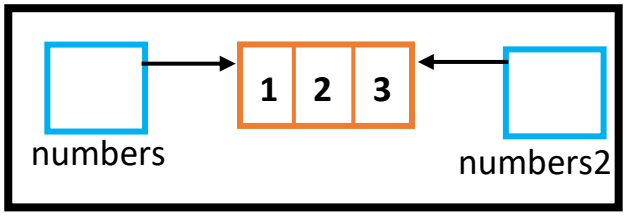
```
Integer count1 = 3;
Integer count2 = count1;
```

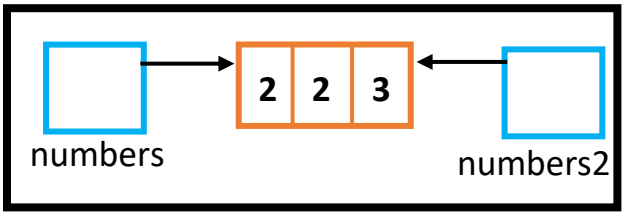


Since the value held in a reference type is a memory address, the memory address is what is copied over. Therefore, now you have two variables that point to the same memory location. Change the value pointed to by count1 and count2 will change as well! Now, to be fair, an Integer is actually an immutable data type, so Java does some different things with it that mess with what happens when the value changes. So, as better proof for what happens when you copy a reference type, consider this:

```
int[] numbers = {1, 2, 3};
int[] numbers2 = numbers;
numbers[0] = 2;
System.out.println(numbers2[0]);
```

Arrays are actually reference types, regardless of what data type they hold. So, an array of ints and an array of Integers are both reference types since they are both arrays. Since an array is a reference type and since reference types copy over the memory address, changing the element of one will change the element of the other:






```
int[] numbers = { 1, 2, 3 };
int[] numbers2 = numbers;
```

```
numbers[0] = 2;
```

So, to recap:

Value Type	Reference Type
Is just a value and nothing else	Is an object of an entire class
Holds the value itself	Holds a memory address to the value
Doesn't have any methods	Has methods
Cannot be set to null	Can be set to null
Can't be used as a generic type parameter	Can be used as a generic type parameter
When copied, value is copied over	When copied, memory location is copied over

In addition to Integer, Float, Double, Character and Boolean, other reference types include String, ArrayList, HashMap, arrays of any data type, and any class you create or any class Java comes with.

Ternary operator

Take a look at the following code:

```
Scanner scanner = new Scanner(System.in);
int x;
int y = scanner.nextInt();

if (y == 0)
  x = 0;
else
  x = 1;
```

This grabs the value of y from the user and examines it. If the value is 0, then x becomes 0, otherwise x becomes 1. A shorter way of writing this is to use the ternary operator.

```
Scanner scanner = new Scanner(System.in);
int x;
int y = scanner.nextInt();

x = (y == 0 ? 0 : 1);
```

The question mark goes after the condition. If the condition is true, the first value is returned, which is 0. If the condition is false, the second value is returned, which is 1. The code is identical to the first but is a nice shorthand that is used decently often and is supported in a wide variety of languages.

Interfaces

An interface is a data type that acts similar to a class, however all of its methods are abstract. It is not possible for the methods to have a body. It is also capable of having fields, however the fields must contain constant values (i.e. the field's value can never be changed once it is set). Take a look at the following interface:

```
interface IAdd {
  int add(int x, int y);
}
```

This is an interface that only has a single method in it. We call these "functional interfaces" and they will be important for a later discussion on lambda functions. For now, just focus on the fact that our IAdd interface has a method called add that receives two ints and produces an int.
I want to add two numbers together using this interface. There are actually 3 different ways I can do this.

Approach 1: Create a new class

I could easily just create my own class that implements this interface. I would then create the code for the method. This would look like the following:

```
class Add implements IAdd {
  public int add(int x, int y) {
    return x + y;
  }
}
```

I have a class called Add that implements the interface called IAdd. I can now instantiate this class, and call on its add method:

```
IAdd iadd = new Add();
System.out.println(iadd.add(5, 3));
```

Approach 2: Use an anonymous class

Java gives us the ability to specify the body of the class, without specifying its name. Doing things this way cuts down on the amount of code we need to write and allows us to write the code inline with the rest of our logic. Here is an example:

```
IAdd iadd2 = new IAdd() {
  public int add(int x, int y) {
    return x + y;
  }
};
System.out.println(iadd2.add(5, 3));
```

It is important to note that this does not indicate we have somehow instantiated an interface. That is not possible! Instead, Java (well actually javac) creates a class for us, tags that class as implementing the interface IAdd, and then adds the body we specified above into the new class. That body better implement all the

methods in the interface or our code will not compile. If we want, we can add the @Override annotation to help us catch any mistakes when overriding the add method.

Approach 3: Lambdas

Another approach we could use are lambda functions. Lambda functions are functions that have no name. Similar to approach 2 above, if we create a function without a name, Java will create the class and the function for us, and then attach the body of the lambda function to the newly created function. Let's see that in action:

```
IAdd iadd3 = (x,y) -> x + y;
System.out.println(iadd3.add(5, 3));
```

Believe it or not, that little amount of code does the same as the other 2 approaches above! Let's unpack what is going on here. First, Java sees that we have a variable, iadd3, that can hold any object that implements the IAdd interface. When Java sees the signs of a lambda function (the lambda operator -> gives it away), it knows that it will be given the parameters and the body of a function. Which function you ask? Well the add function of course. And just how does it know the add function is what we are specifying? That's because there is only a single function in our interface. To properly use lambda functions we must have a functional interface, which, as explained earlier, only has a single function. Otherwise Java would have no clue which function we are giving it the body to. The first part of the lambda function is the parameters. Notice how we don't have any data types. That's because the add function has already specified that they must be ints. We can add the word int if we want to but this isn't necessary. After the -> operator we have the body of the function. This specifies what the function returns, which is simply the addition of x and y. That's it! Java now has everything it needs to create a class, create an add method inside that class, and give that add method the body we specified.

We could also have written it like this:

```
IAdd iadd3 = (x,y) -> { return x + y; };
System.out.println(iadd3.add(5, 3));
```

With lambda functions, if you do <u>not</u> put braces around the body, Java will evaluate the entire body as an expression and automatically return the result of that expression. This however comes at a cost since if we want to do more interesting things and/or have more than one line of code in our body, this won't work. So by putting braces, we can have more lines of code. This means we can either change the lambda function to what you see above (with the braces), or even do something like this:

```
IAdd iadd3 = (x,y) -> {
    int z = x + y;
    return z;
};
System.out.println(iadd3.add(5, 3));
```

But, remember, if we do put braces, Java now has to be told what to return, so we must use the return keyword like a normal function. So to summarize:

No braces around the body	Only allows a single line that is automatically evaluated and returned
Put braces around the body	Can have more lines of code in the body but must treat it like a normal function and use the "return" keyword if you are needing to return a value.

Of course if the function you are creating doesn't return a value (i.e. is a void function), then the return keyword isn't used as you aren't returning anything. So, for example, if the interface is:

```
interface IPrintHelper {
  void printSomething(String msg);
}
```

You can use a lambda function like so:

```
IPrintHelper iPrint = (msg) -> { System.out.println(msg); };
iPrint.printSomething("hello");
```

Note the two semi colons at the end. This is intention as the first semicolon ends the line that prints (this is a normal function body since we added the braces, therefore statements should end in a semicolon). The second semicolon is for the entire line that contains the lambda. This may make more sense if we formatted the line differently:

```
IPrintHelper iPrint = (msg) -> {
  System.out.println(msg);
};
```

We could also omit the braces. Since the System.out.println function doesn't produce a value, Java shouldn't be confused by the lack of braces. In other words, if we don't use braces and if the body of the lambda function produces a value, Java will attempt to return it. If the body doesn't produce a value, then nothing is returned. So we could write the above code without braces like this:

```
IPrintHelper iPrint = (msg) -> System.out.println(msg);
```

Since you will be using lambda functions in one of your programming assignments, I feel like we should talk about one more variation, just to make sure you understand this concept well. What happens if the function doesn't take any parameters. Imagine we had the following functional interface:

```
interface IPrintHello {
  void printHello();
}
```

In this case we could use the following:

```
IPrintHello iPrintHello = () -> {
  System.out.println("hello");
};
iPrintHello.printHello();
```

Note that just like with a normal function, even when we don't have parameters we still must use parentheses.

Lambda functions are a very powerful tool. They are so useful that these days most of our modern languages we use have them. Java, C++, C#, and Python (just to name a few) all support lambda functions, even though the syntax might differ (for example, Python literally has the keyword "lambda"). To learn more, including some more use cases of lambda functions, here is a great video on them: https://youtu.be/tj5sLSFjVj4 .

Variable arguments

Java supports a feature known as variable arguments or "varargs". What this means is that we can send as many arguments to a function as we like. Let's use this concept to modify our existing code. First we will change our interface:

```
public interface IAdd {
  int add(int... nums);
}
```

Notice the ellipsis or "..." operator. This tells Java to take in as many arguments as we give it. They do however all have to be ints as denoted by the word "int" before the ellipsis. Finally, the parameter nums receives all the arguments as an array. Now let's revisit our three approaches.

Approach 1 turns into the following:

```
public class Add implements IAdd {
  public int add(int... nums) {
    int sum = 0;
    for(int i = 0; i < nums.length; i++){
      sum += nums[i];
    }
    return sum;
  }
}
```

Notice the use of a local variable "sum" to add up all the numbers. Also notice how we use Java's for loop to iterate over the integers passed into the function. Remember that "nums" is just an array, so accessing values via `[]` and checking its size by using .length works just fine.

Approach 2 turns into the following:

```
IAdd iadd2 = new IAdd() {
  public int add(int ... nums) {
    int sum = 0;
    for(int num : nums) {
      sum += num;
    }
    return sum;
  }
};
System.out.println(iadd2.add(5, 3));
```

We could have done the same thing as approach 1 above, but instead let's use this as an opportunity to learn about the foreach loop.

Foreach loop

Java's foreach loop (also known as an enhanced for loop or a range-based for loop) is a special type of loop that's more akin to Python's for loop. It naturally iterates over a collection with less work than the basic Java for loop. If you look at the foreach loop above, you'll notice that first we specify a variable that will be equal to each of our values as we iterate through the array. We called this variable num in the code above. By convention we use a plural name for an array and a singular name for the 'for loop' variable. We read this for loop as "for each integer num in the array nums". Each iteration we simply add the value of num to our sum variable. When we are finished, we return the value of sum.

Approach 3 needs a little work. We can't just add into the code like we have done for the other approaches. This is because by default, the lambda function only allows a single line of code and whatever that line returns becomes the returned value of the function. If we want to have more than one line of code, we must add braces. Approach 3 becomes:

```
IAdd iadd3 = nums -> {
  int sum = 0;
  for(int num : nums){
    sum += num;
  }
  return sum;
};
System.out.println(iadd3.add(5, 3));
```

Notice the braces after the lambda operator. This allows us to have more lines of code. When we use this, we must specify the keyword "return" to show what is returned, unlike before where we omitted the return keyword. Also notice that I don't have parenthesis around the parameter nums. You can add parenthesis if you want but these are optional.

Javadocs

We can use normal comments if we want but Java comes equipped with a special form of comment known as the Javadoc comment. This is used to describe any user defined type (classes, interfaces, etc.) and can also be used on fields and methods. Let's look at the following code:

```
/** Class that implements IAdd */
public class Add implements IAdd {
  /** implementation of IAdd's add method
      @param nums numbers to be added
      @return sum of all numbers
  */
  public int add(int... nums) {
    int sum = 0;
    for(int i = 0; i < nums.length; i++){
      sum += nums[i];
    }
    return sum;
  }
}
```

This is our Add class that implements the IAdd interface. The comment above the class is a Javadoc comment. It starts with a /** and ends with a */. It contains a description of what our class does and what the purpose of our class is. Notice the Javadoc comment above the add method. In addition to the description, it contains the annotations @param and @return. @param is used for parameters. You state the name of the parameter followed by a description of that parameter, separated by spaces (i.e. @param SPACE nameOfParam SPACE description of param). Each parameter should be on its own line. @return allows you to specify what the return value is. This isn't for you to simply specify the data type returned, as that information is clear from the method header. Instead, this is for you to specify the meaning of the value being returned.
There are other annotations you can use. For more information, look at the wiki article from this link: https://en.wikipedia.org/wiki/Javadoc . It is important to familiarize yourself with using Javadocs.

To generate the documentation files from your Javadocs, simply run the javadoc tool on the command line:

```
javadoc nameOfFile.java
```

This will generate several files. Opening the index.html file in your web browser will allow you to view the documentation.

Parameter passing

Java uses something called "pass by value" when passing arguments to a function. This means that if you use a variable, the value of that variable is passed. The value of a value type variable is the actual data. The value of a reference type is the memory address it holds. This is actually very efficient as it means that to pass large objects and large arrays, Java only passes the memory location of these things! Have an array with 10,000 elements? Java passes a single memory address. Have an object of a class with 10,000 instance variables? Java

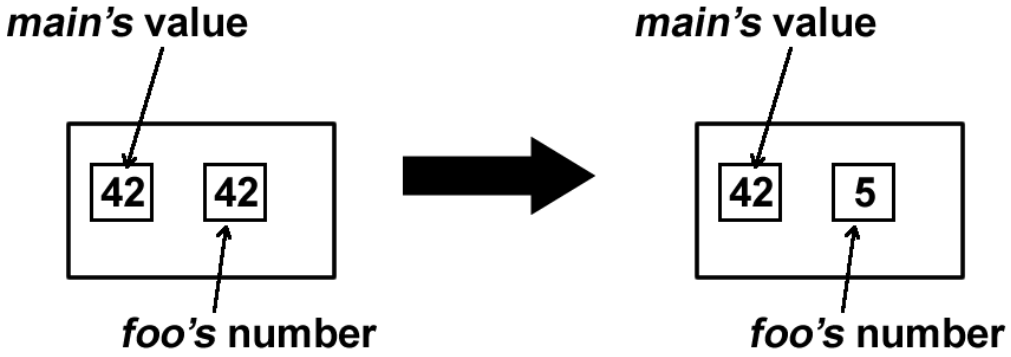
passes a single memory address. Have an int? Java passes that int. Simple. But it does lead to some interesting side effects, though.

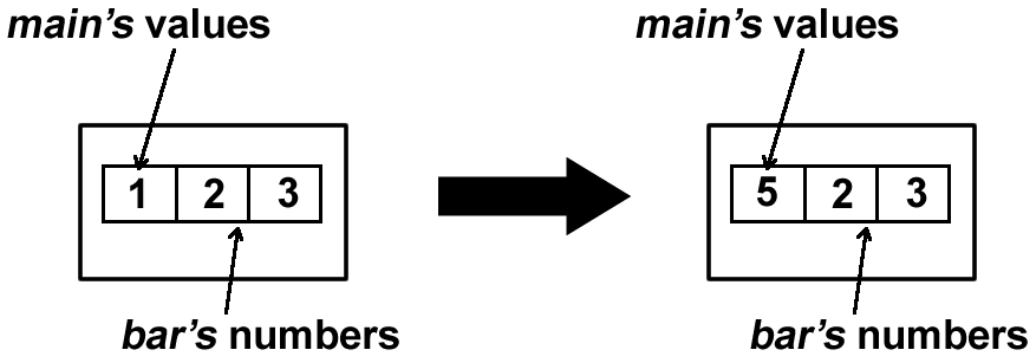
Consider the following code:

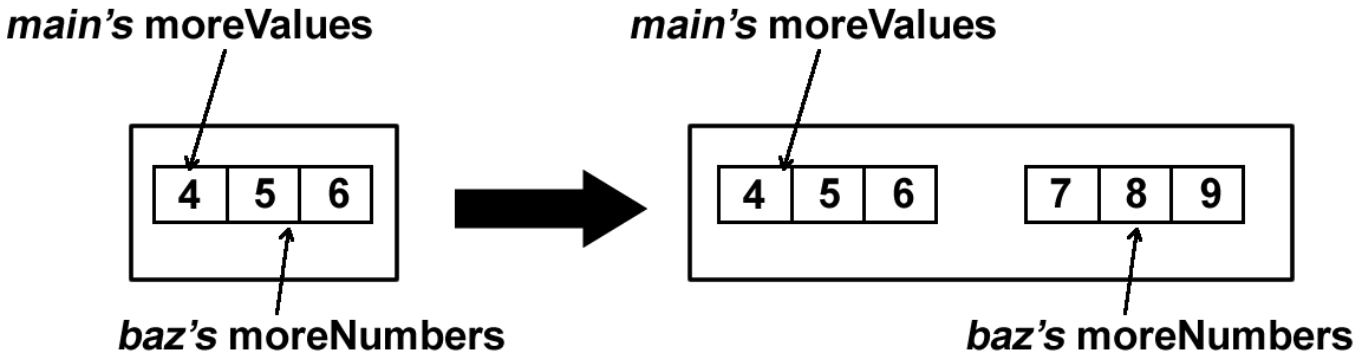
```
public class PassByValueExample {
  public static void main(String[] args) {
    int value = 42;
    foo(value);
    System.out.println(value);

    int[] values = { 1, 2, 3 };
    bar(values);
    System.out.println(values[0]);

    int[] moreValues = { 4, 5, 6 };
    baz(moreValues);
    System.out.println(moreValues[0]);
  }

  private static void foo(int number) {
    number = 5;
  }

  private static void bar(int[] numbers) {
    numbers[0] = 5;
  }

  private static void baz(int[] moreNumbers) {
    moreNumbers = new int[] { 7, 8, 9 };
  }
}
```

If you execute the code, you might find something surprising. This is something you may have learned in previous classes (or not!). When we pass the variable “value” to the function foo, the data inside of it is copied over. In other words, the number 42 is sent to the function. Changing this while inside of the function does not alter the “value” variable in main at all. However, when calling bar, we don’t pass all the values in the “values” array, instead, we pass the memory location. After all, an array is a reference type so we pass the memory location of where the array is stored in memory. Any alterations to this array, will alter the “values” array in main as well, since both the “values” variable in main and the “numbers” variable in bar both point to the same array. But why doesn’t the same thing happen in baz? This is because baz changes which array the parameter “moreNumbers” points to. If you make it point to another array, then changes will only happen to that array and not to the original array (i.e. the array that “moreValues” in main points to remains unchanged).

This diagram might help explain these concepts:







Enums

This is something you may or may not have encountered in a past course. Enums are a special construct that allow you to not only create a new data type, but actually create all the valid values of that data type! Imagine you need a number, however only the values between 0 and 3 are actually used. Perhaps this signifies one of 4 choices in a menu. If you use an int to represent which choice the user selects, you could have an invalid value. For example, the int may be 4, or 5, or 100, or -1. All of these would be invalid. Instead, what if you could develop a new data type where the only valid values are 0, 1, 2 and 3. You could use an enum to do just this:

```
enum MenuChoice { ZERO, ONE, TWO, THREE }

public class EnumTest {
  public static void main(String[] args) {
    MenuChoice choice = MenuChoice.ONE;
    System.out.println(choice);
  }
}
```

The enum we created called “MenuChoice” becomes a data type. This data type only has 4 valid values (i.e. ZERO, ONE, TWO and THREE). When we create a variable of this data type, that variable can only be set to either MenuChoice.ZERO, MenuChoice.ONE, MenuChoice.TWO, or MenuChoice.THREE. But what if we want to actually see the numeric value behind these choices? If we print out the choice variable, we get the value “ONE” printed on the screen (as a string). What if we try to cast the choice variable to an int, like this:

```
System.out.println((int)choice);
```

Java doesn’t seem to like this and this gives us a compiler error:

```
EnumTest.java:11: error: incompatible types: MenuChoice cannot be
converted to int

    System.out.println((int)choice);
                            ^
```

To see the integer value behind each choice, we will have to give each choice a value manually. We will need to create an instance variable (also called a “field”) and assign an int value for each choice. Here is how we can do this:

```
enum MenuChoice {
  ZERO(0),
  ONE(1),
  TWO(2),
  THREE(3);

  public int value;

  MenuChoice(int value) {
    this.value = value;
  }
}
public class EnumTest {
  public static void main(String[] args) {
    MenuChoice choice = MenuChoice.ONE;
    System.out.println(choice.value);
  }
}
```

Here we have created an int variable called “value” and added it to our enum. We then create a constructor that takes in an int value and sets the field “value” to the parameter “value” that it took in. Finally, we change our creation of enum values from ZERO, ONE, TWO, THREE to ZERO(0), ONE(1), TWO(2), THREE(3) meaning we

instantiate 4 MenuChoice objects named ZERO, ONE, TWO and THREE and then give each one an int value to send to the constructor. To retrieve this value, we simply use ".value" after the object, such as "choice.value" in the code above.

Enums can behave like classes in this way and can have fields just like classes. These fields can be of any data type and can store any information we want to hold for that enum value. However, we still only have a set number of values/instances to use and we can't create new values/instances. So, for example, we can't assign the variable "choice" to something like MenuChoice.FOUR since it doesn't exist in our enum. We are limited just to the values created in our enum. If we want FOUR to exist as a value, we must create it in our enum (e.g. FOUR(4)).

Conclusion

There are many more things to learn in Java. Here are some things to Google search:

- Exception handling using try and catch keywords
    - Runtime vs. Normal exceptions
- Method overloading
    - Python actually can't do this, but Java can. Learn what method overloading is.
    - Hint: it is not the same as operator overloading
- Static type system (which Java has) vs. a dynamic type system (which Python has)
    - The type of static type system that Java has is technically called manifest static typing
    - The type of dynamic type system that Python has is sometimes referred to as "duck typing"

If you want to get a good job in the very competitive computer science industry, don't just do the minimum in your classes. Research the topics in the list above. Learn more and put yourself ahead of others competing for the same position. Also, remember, the more you know the higher you can negotiate your pay, so keep learning!