Module 1-6

Introduction to Objects (via Strings)

Stack vs Heap

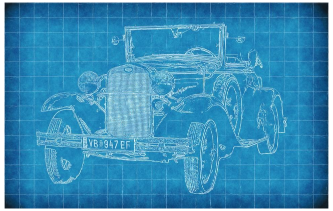
Java will store your data into 2 separate types of spaces: the stack, and the heap.

- The stack is akin to a “stack of items piled one on top of the other in a linear fashion”. It follows the LIFO (Last In First Out) principle.
  - All data of **primitive types** is stored in the stack.
- The heap is a binary tree data structure.
  - **Objects** created in the course of your application are stored in the heap.

Compared to other languages, the management of memory in Java is mostly an automatic process handled by the JVM.

Objects

Objects are instances of classes. For now think of classes like blueprints. Objects have properties (also called members, or data members) and methods.



A **class** is like a blueprint, it is not the thing you're building, but it describes what you're building

From a class, we can create as many objects of that class we need.

Objects: Properties and Methods

Objects have properties (also called members, or data members) and methods.

Consider these vehicles, they were all created from the same blueprint. The blueprint specifies that each vehicle should have a color, **color is therefore a property of the object**.

Objects also have methods. Again, consider some of the things a vehicle can or needs to do: start the engine, go in reverse, check how much fuel it has left. **These are examples of methods a vehicle object might have.**

Objects: Arrays

You have been using two types of objects so far (perhaps unbeknownst to you!), Arrays and Strings. Let's consider Arrays in the context of objects.

- Arrays have a length property: **myArray.length**
- Arrays also have methods:

```
String []  myStringArray  = {"April","Pike","Kirk"};
String []  myOtherArray  = myStringArray;
boolean check = myStringArray.equals(myOtherArray);
System.out.println(check);
```

Note that we are calling a method called equals that expects 1 parameter.

To access an object's properties or methods we use the dot operator as observed above: myArray.length, myStringArray.equals(myOtherArray)

Strings: length method

Unlike arrays, to obtain the length of a string, a method is called. We know this because of the presence of parenthesis.

```
String myString = "Pure Michigan";
int myStringLength = myString.length();
System.out.println(myStringLength);
// The output is 13.
```

- Note that no parameters were taken, nothing goes inside the parenthesis.
- The method’s return is an integer, we can assign it to an integer if needed.

Strings: charAt method

The charAt method for a string returns the character at a given index. The index on a String is similar to that of an Array, namely that it starts at zero.

```
String myString = "Pure Michigan";
char myChar = myString.charAt(1);
System.out.println(myChar);
// The output is u.
```

- Note that charAt takes 1 parameter, the index number indicating the position in the String you want to extract.
- The method’s return is a character.

Strings: indexOf method

The indexOf method returns the starting position of a character or String.

```
String myString = "Pure Michigan";
int position = myString.indexOf('u');
int anotherPosition =
myString.indexOf("Mi");

System.out.println(position); // 1
System.out.println(anotherPosition); // 5
```

- Note that indexOf takes one parameter (what you’re searching for), it can be either a character or String.
- The method’s return is an integer, if nothing is found it will return a -1. If there are multiple matches, it will return the index corresponding the first one.

Strings: substring method

The substring method returns part of a larger string.

```
String myString = "Pure Michigan";
String mySubString = myString.substring(0, 6);
System.out.println(mySubString);
// output: Pure M
```

- Substring requires two parameters, the first is the starting point. The second parameter is a non-inclusive end point (more on this on the next slide).
- It returns a String, so you can assign the output to a String.

Strings: substring method

Just like with arrays, drawing a table of elements or position is a great way to visualize these concepts. Consider the following method call substring(0, **6**)

0	1	2	3	4	5	6	7	8	9	10	11	12
P	u	r	e		M	i	c	h	i	g	a	n

The first parameter is 0, denoting we will start the new String from the 0th position.

The second parameter is the stopping point. The stopping point (6th element) is not included in the final String.

Hence, the output from the previous page is:
Pure M

Strings: mutability

Let’s look at the same example, but print out the original String instead. What do you think is the output now?

```
String myString = "Pure Michigan";
String mySubString = myString.substring(0, 6);
System.out.println(myString);
```

The output will be “Pure Michigan” not “Pure M”!

- Strings are **immutable**, once created they cannot be changed. The result of the substring operation was assigned to a different String but it had no effect on the original String.
- The only way to get a new String value is by re-assigning myString using the = operator to a new value.

Strings: mutability

Here the output is more will be Pure M again:

```
String myString = "Pure Michigan";
myString = myString.substring(0, 6);
System.out.println(myString); // Pure M
```

Compare this to the example on the previous page:

```
String myString = "Pure Michigan";
String mySubString = myString.substring(0, 6);
System.out.println(myString); // Pure Michigan
```

Strings: Other Functions

There are a lot of helpful String methods, some additional ones are listed below, feel free to research their use:

- trim(): removes empty white space from the end of a String.
- toLowerCase(): changes all letters to lowercase.
- toUpperCase(): changes all letters to uppercase.
- replace(): performs a search and replace.

Strings: Comparisons

The proper way to compare Strings is to use the equals() method.

```
String myString = "Pure Michigan";
String myOtherString = "Pure Michigan";
String yetAnotherString = "Ohio so much to discover";

if (myString.equals(myOtherString)) {
        System.out.println("match");
}
```

Do not use == to compare Strings! The results are unpredictable. Strings are objects and == is checking to see if they are the same object, not if they contain the same value.

Strings: Comparisons

The previous discussion on why == should not be used with Strings illustrates an important concept concerning assigning objects (like Strings and Arrays) to variables.

```
String myString = “Hello”;
```

The area to the left of the parenthesis is known as a **reference**.

A reference does not actually store an object, it only tells you where it is in memory.

Strings: Comparisons

One way to think about it is like this: a reference is like a key with a number tag, it does not store anything by itself, but there is a locker with that number on it that holds the actual object.