Module 1-6

Introduction to Objects (via Strings)

Reference vs Primitive Types

- You have now encountered various **primitive data types**: int, double, boolean, float, char, etc.

- We will now discuss **reference types**:
  - You have encountered these already - Arrays and Strings are reference types.
  - Objects that you instantiate from classes that you write are also reference types.

Properties and Methods

Reference types often have properties (also called members, or data members) and methods.

These vehicles were 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 do: start the engine, go in reverse, check how much fuel it has left.

Objects: Arrays

Let's consider Arrays in the context of objects.

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

```
boolean check = myStringArray.equals(myOtherArray);
System.out.println(check);
```

To access an object's properties or methods we use the dot operator as observed above. Methods have a set of parentheses.

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 value is of type char.

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.
- 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

Let’s code!

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";
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 has no bearing on the original String.
- The only way to get a new String value containing the smaller String is by re-assigning myString using the = operator to a new variable.

Strings: mutability

Here is how to get around this:

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

Let’s code!

Strings: Comparisons

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

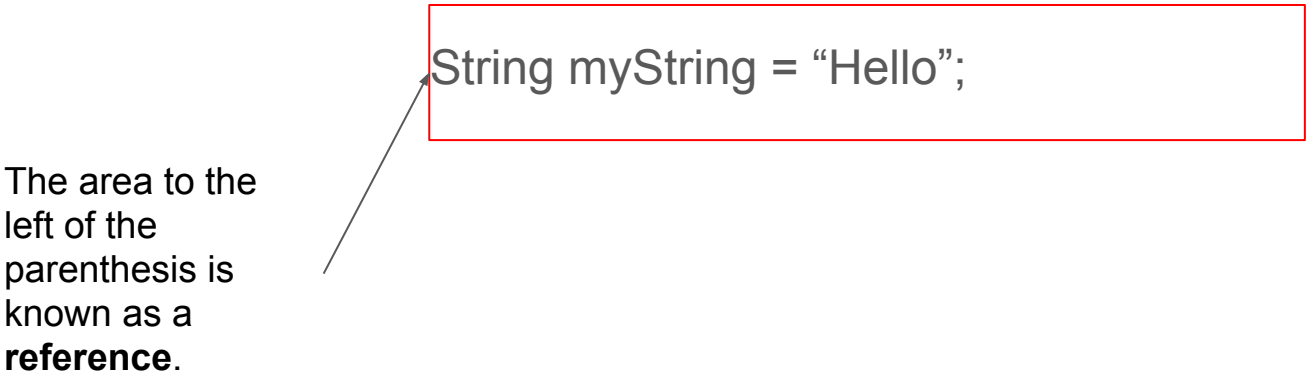
```
String myString = "Pure Michigan";
String myOtherString = "Pure Michigan";
String yetAnotherString = "Ohio so much to discover";

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

Do not use == to compare Strings!

Objects: References

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



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

Objects: Key & Locker Analogy

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. With this analogy, the key with the number 7 is called myString.

“Hello”

The “new” keyword

- Java is built around thousands of “blueprints” called classes and provides you with the ability to create your own classes.
- The **new** keyword is typically used to create an instance of a class.
- We refer to these instances as **objects** of a specific class.
- We have already seen this before, consider the declaration of an array.

```
int [] scores = new int[5];
```

- If we use the concepts we just learned, the above statement means, create a reference (key) of type integer array. Proceed to create a new instance of this array of length 5.
- Strings aren’t required to follow this convention… but if you want to you can:

```
String aString = new String(“Hello”);
```

Null objects

- If a reference type is declared without an equal sign, its value will be **null.**

```
int [] scores;
```

- This is difficult to simulate with the two reference types you know, as the compiler will not allow you to get away with this, we will discuss this in more detail in later modules.