Module 1-6

Introduction to Objects

Module 1 Day 6

Can\Do you?

1. … explain the concept of an object as a programming construct
2. … describe the difference between objects and classes and how they are related
3. … instantiate and use objects
4. … understand the terms Declare, Instantiate and Initialize
5. … understand how objects are stored in RAM
6. … explain how the Stack and Heap are used with objects and primitives
7. … define the terms Value-type and Reference-type
8. … describe the String class, its purpose and use
9. … call methods on an object & explain their return values based on the signature
10. … understand immutability and what that means for handling certain objects
11. … explain object equality and the difference between `==` and `equals()`

Reference vs Primitive Types: The Stack and Heap

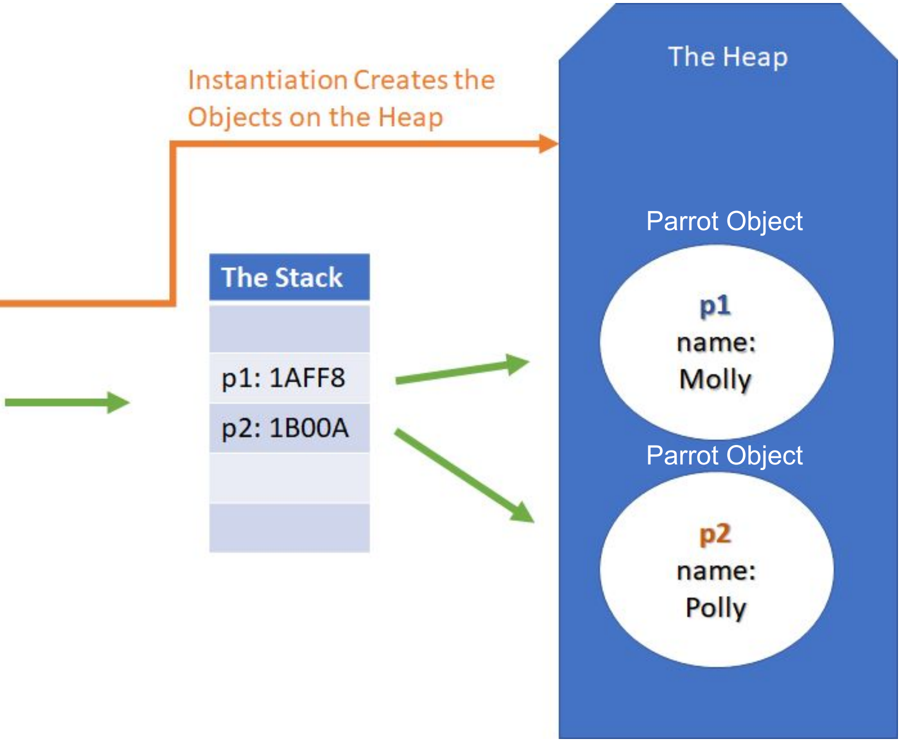
- You have now encountered various **primitive data types**: int, double, boolean, float, char, etc. Primitives exist in memory in containers sized to fit their max values in an area known as the Stack.

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

Objects: References

```
Public class Rescue {
    public void populateBirds(){
        Parrot p1 = new Parrot("Molly");
        Parrot p2 = new Parrot("Polly");
    }
}
```

These declared Parrot **references** do not store an object (or its values). They only store the address of where the objects live in memory on the Heap. These addresses are primitives and are stored on the stack.



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.

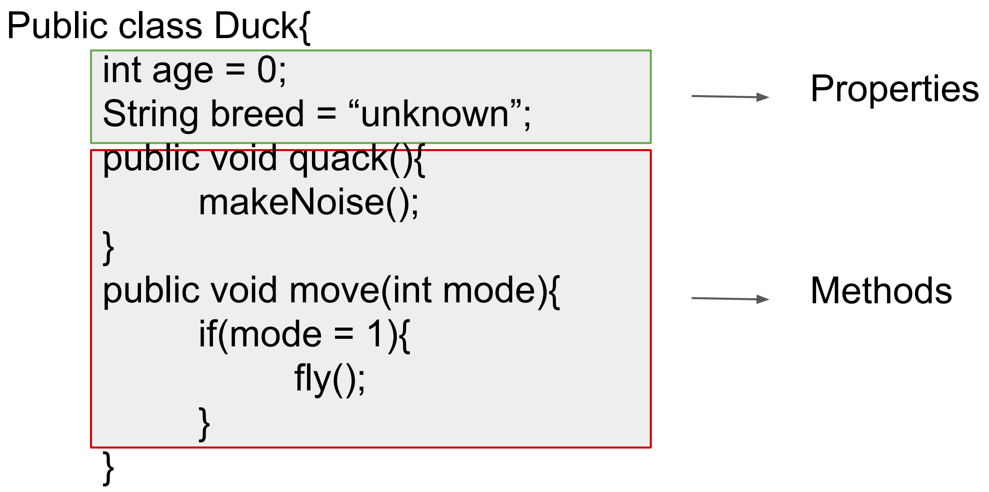
```
String myString = “Hello”;
```

With this analogy, the key with the number 7 is called myString and myString is set to “Hello”

“Hello”

Objects: Properties and Methods

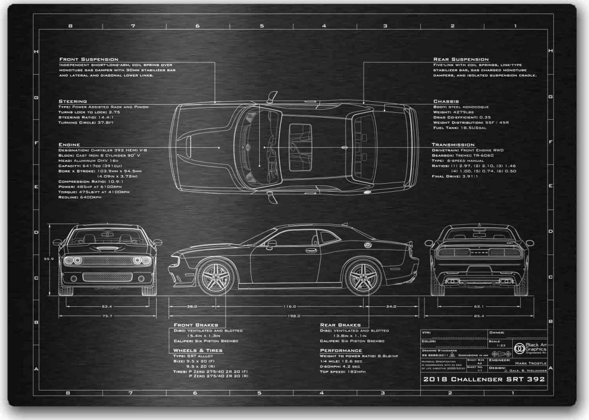
Reference types often have properties (also called members, or data members) and methods. These are commonly thought of as *attributes, or* ***properties*** that define an object's state and ***behaviors***.



```
Public class Duck{
        int age = 0;
        String breed = “unknown”;
        public void quack(){
                makeNoise();
        }
        public void move(int mode){
                if(mode = 1){
                        fly();
                }
        }
```

Objects: Properties and Methods

When defined in code as a Class, these properties and methods form a blueprint for the creation of Objects in memory. A Class is defined by the code, an object is the “physical” manifestation of a ***specific instance*** of that class in memory.



These vehicles were created from the same blueprint. The blueprint specifies that each vehicle should have a color, <u>**color is therefore a property of the object**</u>.

Objects: Instantiation - Creating the Object

- 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];
```

- The statement above create a reference (key) of type integer array, create a new instance of this array with a length of 5.

  P.S. Strings, our intro into the world of objects, aren’t required to follow this convention… but they can:

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

Objects: Null - Lack of Initialization

- 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.

Objects: Arrays

Let’s consider Arrays in the context of objects.

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

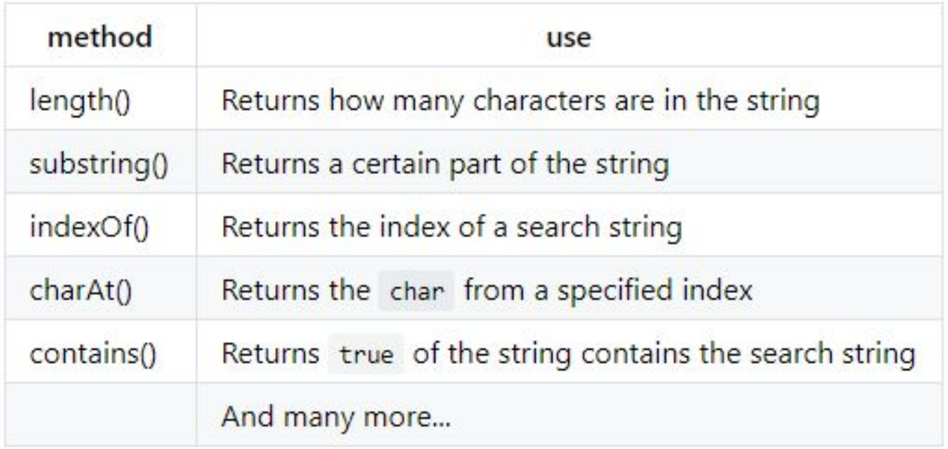
```
boolean check = myStringArray.equals(myOtherArray);
System.out.println(check);
```

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

Break!

Objects: Strings

Like all objects, strings have methods:



method	use
length()	Returns how many characters are in the string
substring()	Returns a certain part of the string
indexOf()	Returns the index of a search string
charAt()	Returns the `char` from a specified index
contains()	Returns `true` of the string contains the search string
	And many more...

To access an object’s properties or methods we use the dot (.) operator, also known as a period. All Methods have a set of parentheses, properties will not.

Strings: Immutable

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
```

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

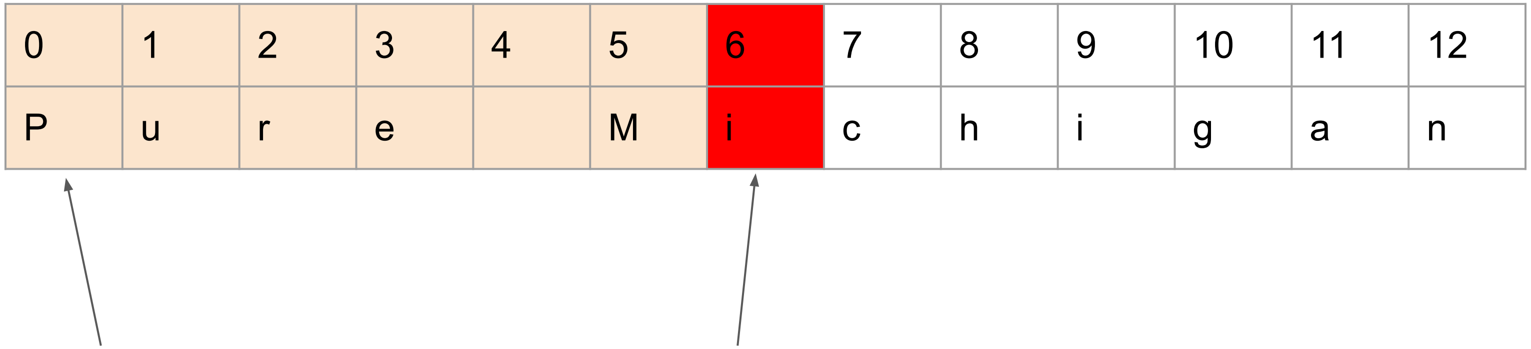
The substring method returns part of a larger string.

```
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

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!