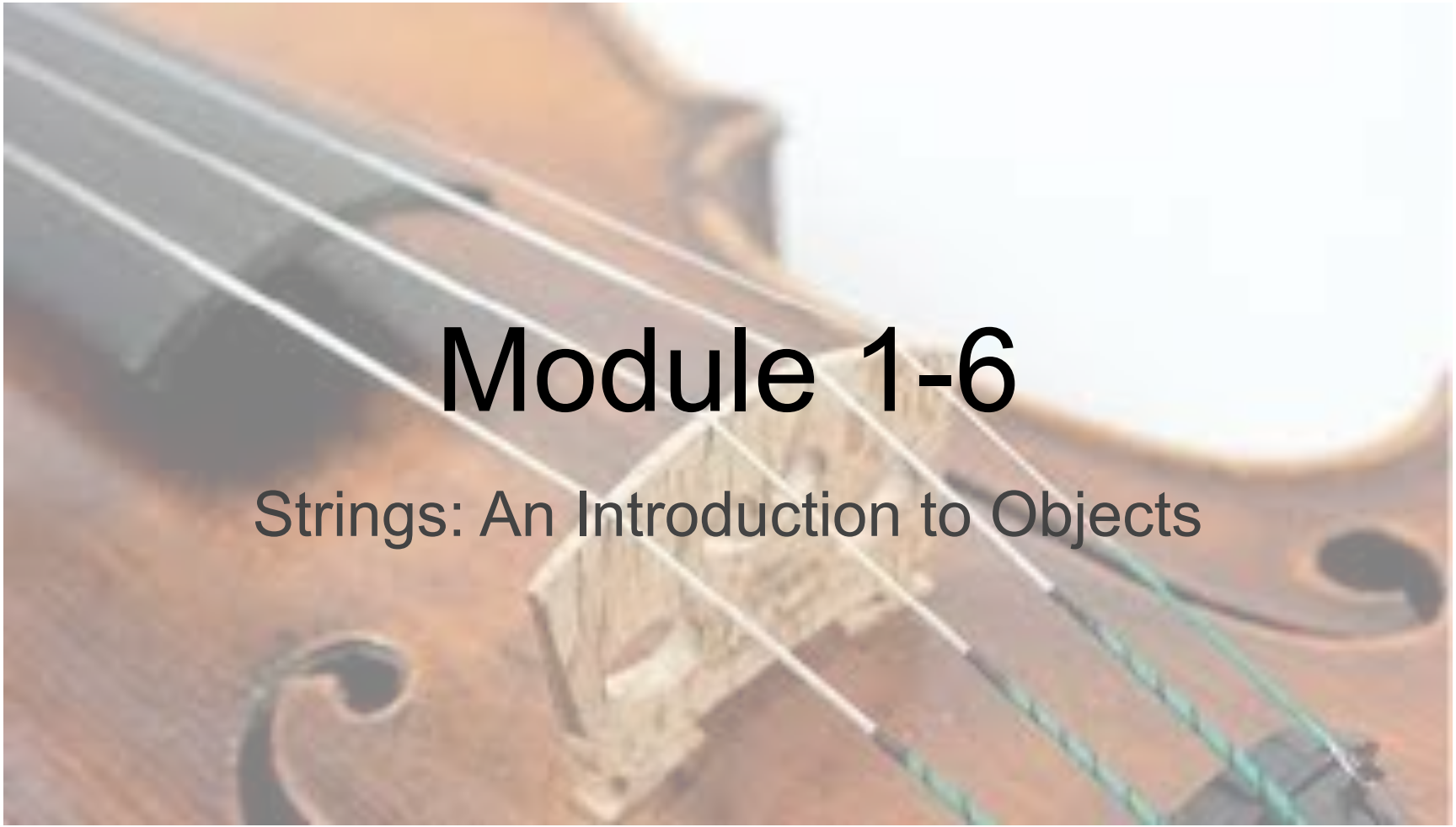


Module 1-6

Strings: An Introduction to Objects

Module 1 Day 6

Can\Do you?

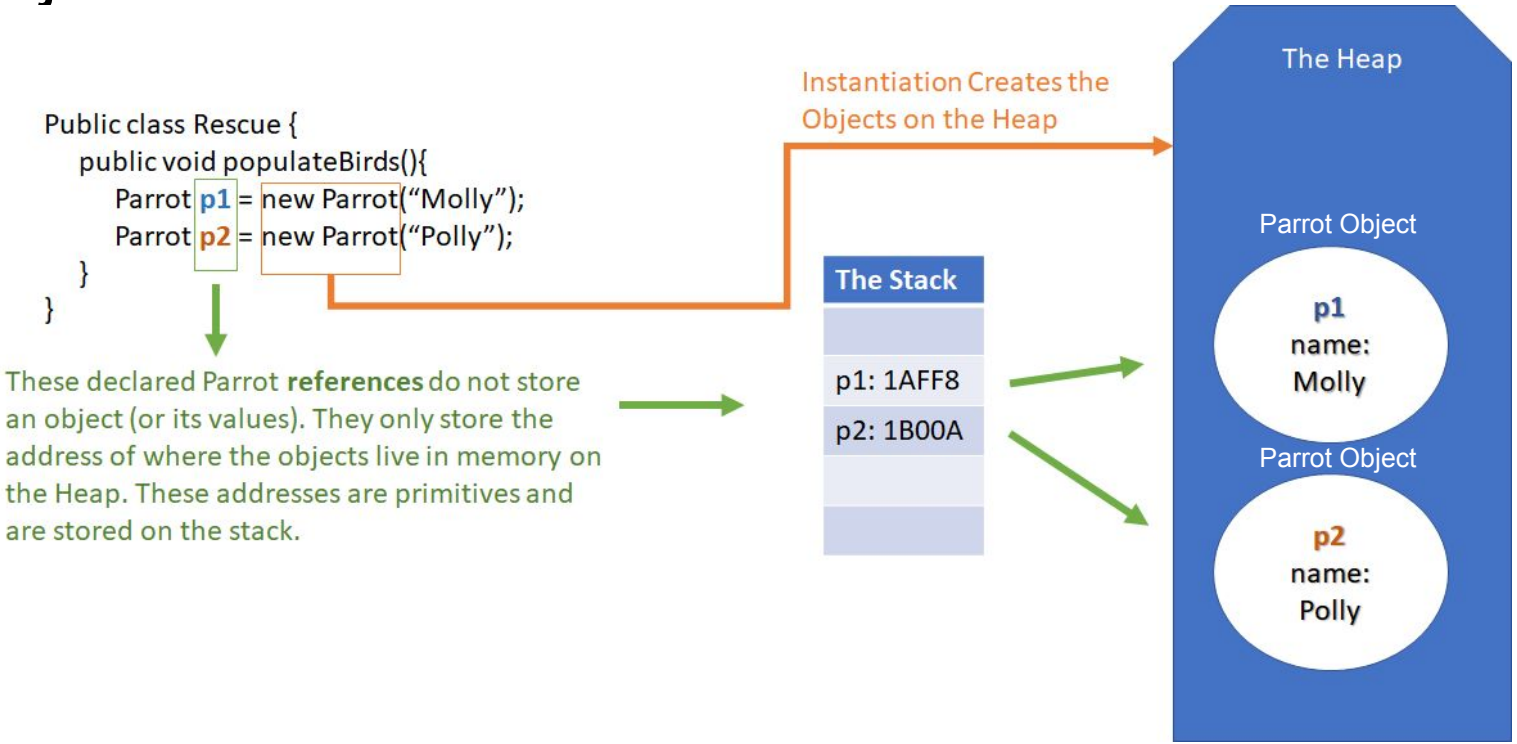
1. … explain the concept of an object as a programming construct
2. … describe the difference between objects and classes and how they are related
3. … instantiate and use objects
4. … understand the terms Declare, Instantiate and Initialize
5. … understand how objects are stored in RAM
6. … explain 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



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

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

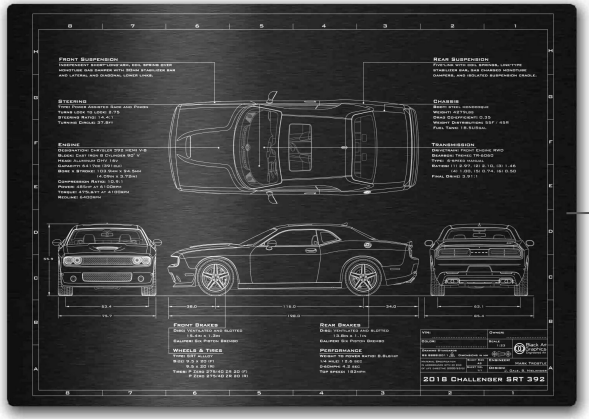
→ Properties

→ Methods

Objects: Properties and Methods

Defined using code as ***Classes***, 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.

CLASS



OBJECTS

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 and can be set once instantiated**.

Objects: Instantiation - Creating the Object

- Java is built around hundreds of these “blueprints” called classes that give us the ability to create in-memory objects in our applications.
- The **new** keyword is typically used to create an instance of a class.
- We refer to these **instances** of classes as **objects** of a specific class.
- This has already been shown to you while working with arrays: consider the declaration of this int(eger) array.

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

- The statement above creates a reference, *scores*, of type integer array, as a ***new*** instance of an array class whose length is 5.

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

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

Objects: Null is a Lack of Initialization

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

```
int[] scores;
```

- This state is difficult to simulate with the two reference types we have discussed up until now.

- We will discuss **null** and the Null Reference Exception in more detail in later lectures.

Objects: Arrays

Let's consider Arrays in the context of objects.

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

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

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

Why would we use ***.equals()*** instead of == ?

Break!

Objects: Strings

Like all objects, strings also have methods. We've seen some, some are new:

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... Oracle Java Docs String Reference

This is just a partial list, to access an object's properties or methods we use the dot (.) operator, aka period. All Methods are referred to using a set of parentheses, properties are not.

Strings: Strings are Immutable

Let's look at the substring method, substring(int start, int end) will return the characters from the string in the specified range:

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

> Pure P
> _______?
```

- What is the value of myString after this code has run?

Strings: Strings are Immutable

Let’s look at the same example, but print out the original String instead.

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

- Strings are **immutable**, once created they cannot be changed, although they may be assigned new values. In other words ***the result of the substring() method has no bearing on the original String.***

- The only way to capture the new String value returned by substring() method for subsequent process is assigning the result of substring() to a new String object using the = operator.

Strings: mutability through assignment

Here is one way to get around this immutable nature of strings:

```
String myString = "Pure Philly";
myString = myString.substring(0, 6);
System.out.println(myString);
> Pure P
```

Strings: length method

Unlike arrays, the length of a string is found using a method. We know String.length() is a method because of the presence of parenthesis.

```
String myString = "Pure Philly";
int myStringLength = myString.length();
System.out.println(myStringLength);
> 13
```

- Note that length() needs no parameters, nothing goes inside the parenthesis.
- The method's return type is an integer and we can assign it to an integer if desired.

Strings: charAt method

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

```
String myString = "Pure Philly";
char myChar = myString.charAt(1);
System.out.println(myChar);
>  u
```

- Note that charAt takes 1 parameter, the index number indicating the position of the character you want to extract from the String.
- 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 Philly";
int position = myString.indexOf('u');
int anotherPosition = myString.indexOf("Phi");

System.out.println(position);
System.out.println(anotherPosition);

> 1
> 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

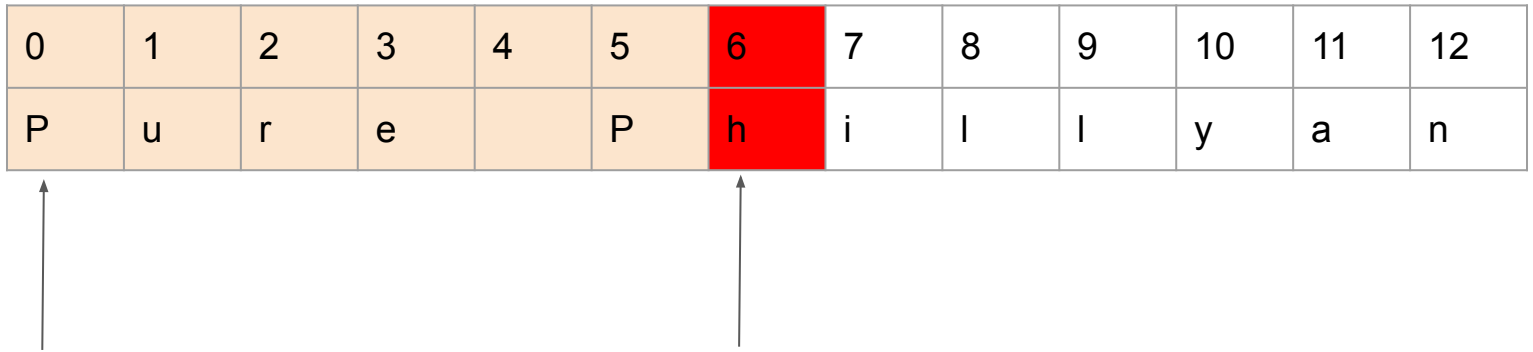
The substring method returns part of a larger string.

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

- 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**)



The first parameter is 0, it sets the start the new String at the 0th position.

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

The output is:
Pure P

Strings: Comparisons

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

```
String myString = "Pure Philly";
String myOtherString = "Pure";
myOtherString = myOtherString + " Philly";

if (myString.equals(myOtherString)) {
        System.out.println("match");
        // What happens with == ?
}
```

Do not use == to compare Strings!