Introduction To Objects

Today's Objectives

- Objects as a programming construct.
- Differences between objects and classes and how they are related.
- Instantiating and using objects.
- Declare, Instantiate and Initialize.
- How objects are stored in memory.
- How the Stack and Heap are used with objects and primitives
- Value-type and Reference-type
- String class: what it is and how it is used.
- Calling methods of an object & understanding their return values based on the signature.
- Immutability: what it is and what that means for handling certain objects
- 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.
    - The ***stack*** is a region of computer memory that manages temporary variables created by each method (remember scope?).
- 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.
    - Reference types are stored on the ***heap***, not the stack.

What is an Object?

An ***object*** is an in-memory data structure that combines state and behavior into a usable and useful abstraction.

- An object lives in memory and each object is different and separate from every other object in our program.

What is a Class?

We don't technically write objects in our code. Objects only exist when our code is running because an object is an in-memory data structure. In order to make objects, we have to write classes.

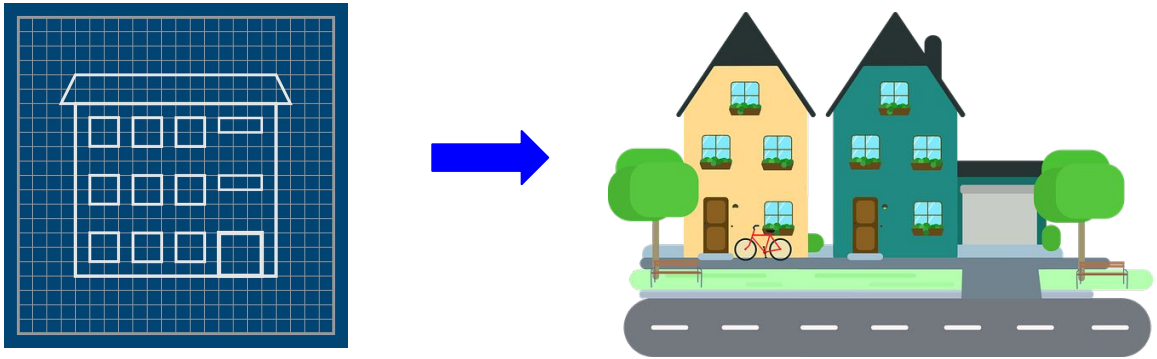
A ***class*** is a grouping of variables and methods in a source code file that we can generate objects out of.

- A class is to an object like a blueprint is to a house. A class defines what an object will be like once the object is created.

- We can even create our own classes, but we'll talk more about that later in the week.

Objects vs. Classes

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 definition.



The blueprint on the left was used to build the two houses on the right. The blueprint specifies that the houses will have a color but the actual color can be different for each house built, so **color is a property which each house has its own version of.** You can think of a class as the blueprint and objects as the actual houses that are built from the blueprint using their own properties, such as color.

Primitive Types vs. Object References

Primitive Types

```
public void createData() {
      int age = 40;
      char grade = 'A';
}
```

stack

age	40
grade	A

Primitive Types vs. Object References

Primitive Types

```
public void createData() {
      int age = 40;
      char grade = 'A';
}
```

stack

age	40
grade	A

Object (Reference Type)

```
public static void createBooks() {
      Book firstBook = new Book("Fly Eagles, Fly");
      Book secondBook = new Book("Gritty's Big Day");
}
```

Primitive Types vs. Object References

Primitive Types

```
public void createData() {
      int age = 40;
      char grade = 'A';
}
```

stack

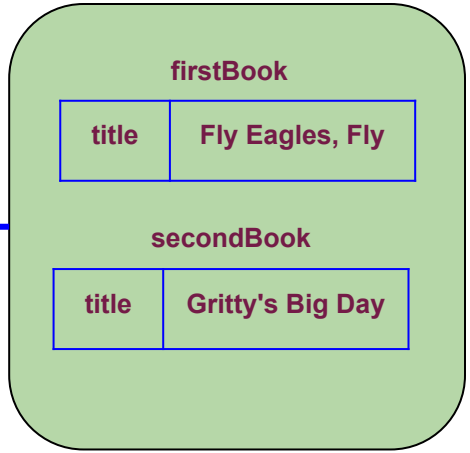
age	40
grade	A

Object (Reference Type)

```
public static void createBooks() {
      Book firstBook = new Book("Fly Eagles, Fly");
      Book secondBook = new Book("Gritty's Big Day");
}
```

instantiation creates objects on the heap

heap



Primitive Types vs. Object References

Primitive Types

```
public void createData() {
      int age = 40;
      char grade = 'A';
}
```

stack

age	40
grade	A

Object (Reference Type)

```
public static void createBooks() {
      Book firstBook = new Book("Fly Eagles, Fly");
      Book secondBook = new Book("Gritty's Big Day");
}
```

instantiation creates objects on the heap

stack

firstBook	1AFF8
secondBook	1B00A

stack holds address where object exists in the heap

heap

firstBook

title	Fly Eagles, Fly

secondBook

title	Gritty's Big Day

Pass By Value vs. Pass By Reference

When passing variables around, primitive variables behave differently than variables that reference memory on the heap.

- If a primitive variable is passed to a method and is modified in the method, the value of the original variable will remain the same after the method call.

- If a reference variable is passed to a method and is modified in the method, the value of the original variable will have the modified value after the call. This is because the variable is referring to a location in memory, so the data is retained even after the method call is removed from the stack.

Primitive Types vs. Object References

Primitive Types

```
public void createData() {
      int age = 40;
      char grade = 'A';
}
```

stack

age	40
grade	A

Object (Reference Type)

```
public static void createBooks() {
      Book firstBook = new Book("Fly Eagles, Fly");
      Book secondBook = new Book("Gritty's Big Day");
}
```

instantiation creates objects on the heap

stack

firstBook	1AFF8
secondBook	1B00A

stack holds address where object exists in the heap

heap

firstBook

title	Fly Eagles, Fly

secondBook

title	Gritty's Big Day

Object References: 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.

The key is the like the variable that holds the memory location.

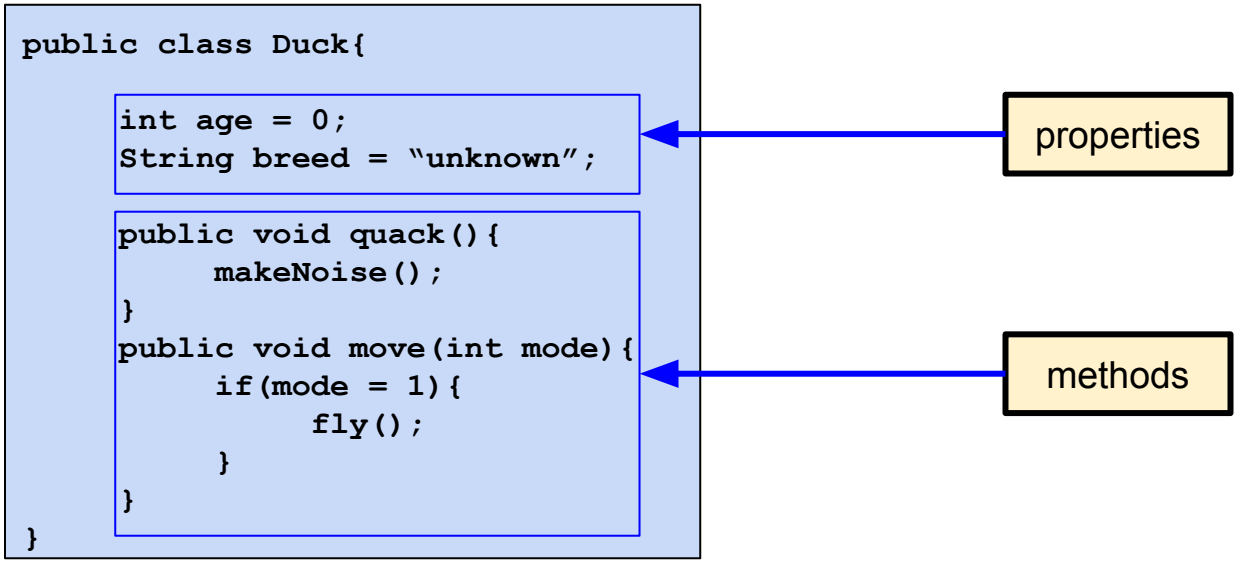
The locker is the memory location the variable points to

```
"Hello"
```

The content referenced by the variable is stored in memory, like things are stored in the locker.

Object Properties and Methods

Objects 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.



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 creates a reference (think: *key*) of type integer array which refers to a new instance of an integer array (think: *locker*) with a length of 5 (think: *locker content*).

- 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");
```

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.

Arrays as Objects

Let's consider Arrays in the context of objects:

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

```
String check = myStringArray.toString();
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

Like all objects, strings have methods. Here are some examples:

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 as observed above. Methods have a set of parentheses.

String Methods: length

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

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

- Note that length() takes no parameters - nothing goes inside the parentheses.
- The method's return is an integer, we can assign it to an integer if needed.

String Methods: charAt

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.

String Methods: indexOf

The indexOf method returns the index of the first occurrence 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. Remember the index is zero-based.

String Methods: substring

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.

String Methods: substring

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
```

There is actually another version of **`substring`** which only requires one parameter. That version starts at the specified index and goes to the end of the String.

- 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.

String Methods: substring

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 position 0..

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

Based on this, the output from the previous page is:

Pure M

Strings Are Immutable

Let's look at the substring example again, 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 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.

Dealing with String Immutability

Here is how to we could update the value in **`myString`**:

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

- When we set **`myString`** to the result of **`substring`**, we are creating a new **`String`** object and replacing the **reference address** of the object in **`myString`**.

String 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");
}
```

YOU SHOULD NOT USE == TO COMPARE STRINGS!!!!!

Introducing: BigDecimal

We can use the BigDecimal class to handle floating point arithmetic correctly.

- The two java primitive types(**`double`** and **`float`**) are floating point numbers, which is stored as a binary representation of a fraction and a exponent.

- The primitive types **`int`** and **`long`** are fixed-point numbers. Unlike fixed point numbers, floating point numbers will most often return an answer with a small error (around 10^-19) This is the reason why we end up with 0.00999999999999998 as the result of 0.04-0.03.

- More info on BigDecimal: https://www.geeksforgeeks.org/bigdecimal-class-java/

Introducing: BigDecimal

`BigDecimal` objects can be created using **`new`** and a parameter such as a **`String`**

```
BigDecimal bigDecimalFromString = new BigDecimal("0.04");
BigDecimal bigDecimalFromDouble = BigDecimal.valueOf(0.03);

BigDecimal difference = bigDecimalFromString.subtract(bigDecimalFromDouble);
```

Introducing: BigDecimal

`BigDecimal` objects can be created using **`new`** and a parameter such as a **`String`**

`BigDecimal` objects can also be created using the **`BigDecimal`** static method **`valueOf`** with a **`double`** value as a parameter. This should be used rather than using **`new`** with a **`double`** parameter to avoid prevision problems inherent in **`doubles`**

```
BigDecimal bigDecimalFromString = new BigDecimal("0.04");
BigDecimal bigDecimalFromDouble = BigDecimal.valueOf(0.03);

BigDecimal difference = bigDecimalFromString.subtract(bigDecimalFromDouble);
```

Introducing: BigDecimal

`BigDecimal` objects can be created using **`new`** and a parameter such as a **`String`**

`BigDecimal` objects can also be created using the **`BigDecimal`** static method **`valueOf`** with a **`double`** value as a parameter. This should be used rather than using **`new`** with a **`double`** parameter to avoid prevision problems inherent in **`doubles`**

```
BigDecimal bigDecimalFromString = new BigDecimal("0.04");
BigDecimal bigDecimalFromDouble = BigDecimal.valueOf(0.03);

BigDecimal difference = bigDecimalFromString.subtract(bigDecimalFromDouble);
```

`BigDecimal` objects perform math operations using object methods such as **`add`**, **`subtract`**, **`multiply`**, **`divide`**, **`pow`**.

Introducing: BigDecimal

`BigDecimal` objects can be created using `new` and a parameter such as a `String`

`BigDecimal` objects can also be created using the `BigDecimal` static method `valueOf` with a `double` value as a parameter. This should be used rather than using `new` with a `double` parameter to avoid prevision problems inherent in `doubles`

```
BigDecimal bigDecimalFromString = new BigDecimal("0.04");
BigDecimal bigDecimalFromDouble = BigDecimal.valueOf(0.03);

BigDecimal difference = bigDecimalFromString.subtract(bigDecimalFromDouble);
```

`BigDecimal` objects are immutable, so if you want to store the result of `BigDecimal` arithmetic, you need to assign it back to a new `BigDecimal` object.

`BigDecimal` objects perform math operations using object methods such as `add`, `subtract`, `multiply`, `divide`, `pow`.

BigDecimal: Division & Multiplication

Division and multiplication with BigDecimal usually involves using a MathContext or a RoundingMode.

```
BigDecimal num1 = new BigDecimal("66.6");
BigDecimal num2 = new BigDecimal("100");

BigDecimal divisionWithRounding = num1.divide(num2, RoundingMode.HALF_DOWN);
BigDecimal divisionaWithContext = num1.divide(num2, MathContext.DECIMAL128);
```

`RoundingMode.HALF_DOWN` rounds towards the "nearest neighbor" unless both neighbors are equidistant, in which case it rounds down. This is known as Banker's Rounding and is commonly used in the U.S.

BigDecimal: Division & Multiplication

Division and multiplication with **`BigDecimal`** usually involves using a **`MathContext`** or a **`RoundingMode`**.

```
BigDecimal num1 = new BigDecimal("66.6");
BigDecimal num2 = new BigDecimal("100");

BigDecimal divisionWithRounding = num1.divide(num2, RoundingMode.HALF_DOWN);
BigDecimal divisionaWithContext = num1.divide(num2, MathContext.DECIMAL128);
```

`RoundingMode.HALF_DOWN` rounds towards the "nearest neighbor" unless both neighbors are equidistant, in which case it rounds down. This is known as Banker's Rounding and is commonly used in the U.S.

`MathContext.DECIMAL128` indicates that the max number of decimal places is 34 and that HALF_DOWN RoundingMode will be used as needed.

BigDecimal: Comparing Values

The **`BigDecimal compareTo`** method returns -1(less than), 0(Equal), 1(greater than) according to values.

- num1.compareTo(num2) > 0 // num 1 is greater than num2
- num1.compareTo(num2) == 0 // num 1 is equal to num2
- num1.compareTo(num2) < 0 // num 1 is less than num2