Unit 2: Using Objects

Overview

Mr. Pelletier, Lord Byng Secondary

Unit 2 Overview

In Unit 2, you'll explore reference data as a way to represent real-world objects in a digital world and discover methods to perform more complex operations.

Topics will include:

- Objects and classes as ways to describe instances, attributes, and behaviors
- Creating objects by calling constructors with and without parameters
- Defining an object's behaviour using methods
- Calling non-static void methods with and without parameters
- Using String objects and methods
- Utilizing class libraries, including Integer, Double, and Math

Lesson 2.1: Objects: Instances of Classes

Mr. Pelletier, Lord Byng Secondary

2.1 Objects: Instances of Classes

Overview

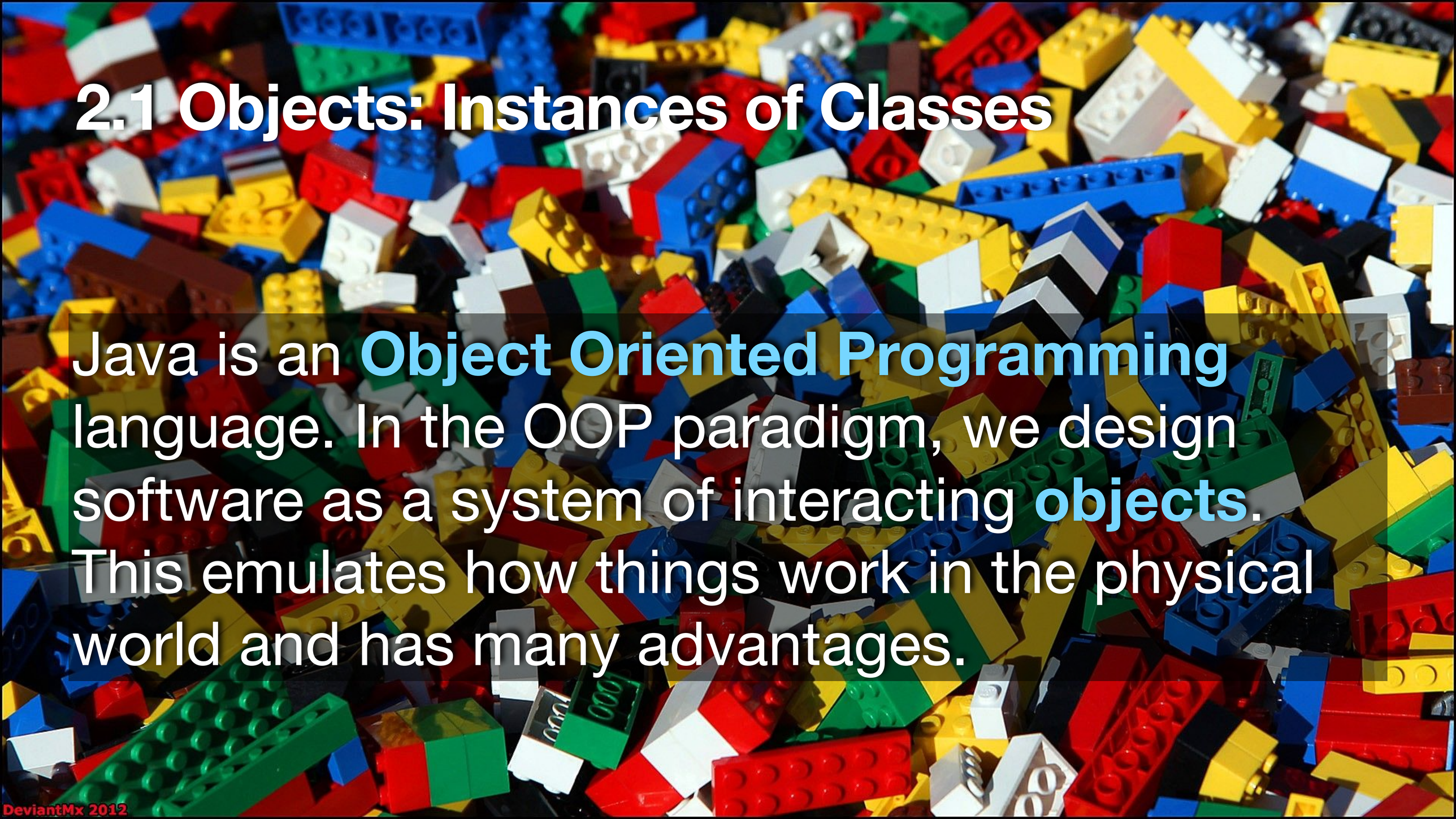
Let’s talk about Object Oriented Programming and classes. We will look at classes in the source code and begin to understand their parts.

We will explore the following terms:

- Object Oriented Programming
- Class
- Object
- Instance
- Attribute
- Instance Variable
- Constructor
- Behaviour Method

2.1 Objects: Instances of Classes

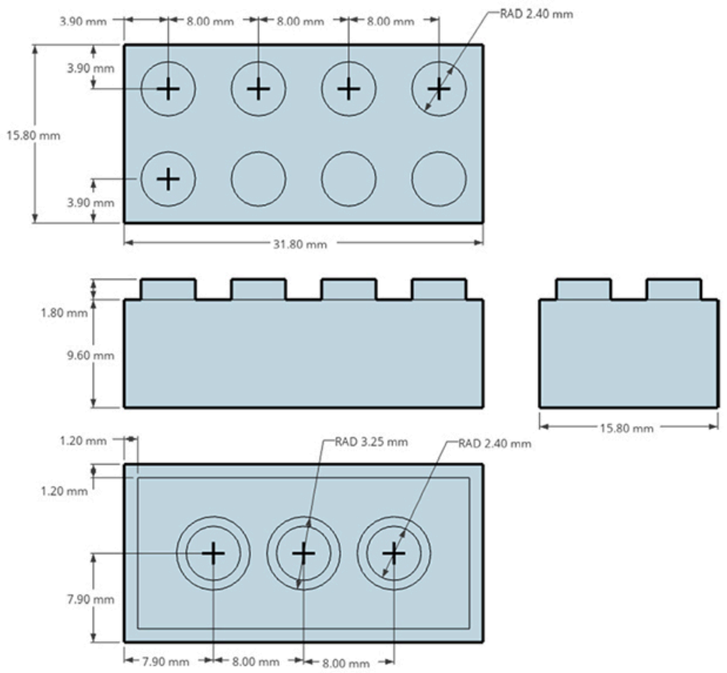
Java is an **Object Oriented Programming** language. In the OOP paradigm, we design software as a system of interacting **objects**. This emulates how things work in the physical world and has many advantages.



2.1 Objects: Instances of Classes

Classes vs. Objects

A **class** is a blueprint for an individual object. It describes the characteristics of objects created from it.



Objects are virtual entities that are created as **instances** of a class. Objects are the building blocks of an OOP program.

2.1 Objects: Instances of Classes

Attributes and Methods

Classes have 3 primary features:

1. **Attributes** describe the characteristics of their objects.
Represented by **Instance Variables**.

2. **Constructors** are special methods that initialize an object's attributes.

3. **Behaviour Methods** are code blocks
that define an object's actions.

2.1 Objects: Instances of Classes

Example: A Dog Class

- What are some attributes of a dog?
- What behaviours does a dog have?
- What are the initial values for a particular dog?

2.1 Objects: Instances of a Class

Class Definition Example

```
class Dog {

  int age;
  String name;

  Dog(int a, String n) {
    age = a;
    name = n;
  }

  void bark() {
    System.out.println(name + " says woof!");
  }
}
```

Attributes are stored in Instance Variables

Constructors initialize attributes

Behaviour Methods allow objects to act and interact

2.1 Objects: Instances of a Class

Class Definition Example

```
class Dog {

  int age;
  String name;

  Dog(int a, String n) {
    age = a;
    name = n;
  }

  void bark() {
    System.out.println(name + " says woof!");
  }
}
```

Note: the constructor name is always the same as the class's name

2.2 Creating and Storing Objects

Creating Instances of Dog Class 1/2

```
class Dog {

  int age;
  String name;

  public Dog(int a, String n) {
    age = a;
    name = n;
  }

  void bark() {
    System.out.println(name + “ says woof!”);
  }
}
```

How do we create a Dog Object?
How do we create an instance of the Dog class?
How do we instantiate Dog?

2.2 Creating and Storing Objects

Creating Instances of Dog Class 2/2

```
class Driver {

  public static void main (String[] args) {

      Dog firstDog, secondDog;

      firstDog  = new Dog(2, “Argyle”);
      secondDog = new Dog(1, “Robespierre”);

      firstDog.bark();
      secondDog.bark();
}
```

Declare Dog variables that can store references to Dog objects

Create instances of Dog objects using the new keyword and calling the constructor!

Now we can make the Dog objects bark

Lesson 2.2: Creating and Storing Objects

Mr. Pelletier, Lord Byng Secondary

2.2: Creating and Storing Objects

Overview 1/2

We will explore the process of **INSTANTIATING** new objects from classes.

We will see that **PARAMETERS** give programmers a way to customize objects.

We will create multiple constructors in one class, leading us to the concept of **OVERLOADING**.

We will learn about storing our new objects with **REFERENCES** to objects and understand the structure of non-primitive variables.

2.2: Creating and Storing Objects

Overview 2/2

We will explore the following terms:

- Formal Parameter
- Actual Parameter / Argument
- The **`new`** keyword
- Signature
- Overloading
- Reference
- The **`null`** keyword
- **`NullPointerException`**

2.2: Creating and Storing Objects

Class Signature

```
class Dog {

  int age;
  String name;

  public Dog(int a, String n) {
    age = a;
    name = n;
  }

  void bark() {
    System.out.println(name + " says woof!");
  }
}
```

This is the Constructor's signature

2.2: Creating and Storing Objects

Formal Parameters

```
class Dog {

  int age;
  String name;

  public Dog(int a, String n) {
    age = a;
    name = n;
  }

  void bark() {
    System.out.println(name + " says woof!");
  }
}
```

These variables are called FORMAL PARAMETERS

These allow you to make constructors more flexible, more customizable. Dog objects can be made with specific ages and names by passing values to the constructor.

2.2: Creating and Storing Objects

Actual Parameters

```
class Driver {

  public static void main (String[] args) {

     Dog firstDog, secondDog;

     firstDog  = new Dog(2, “Argyle”);
     secondDog = new Dog(1, “Robespierre”);

     firstDog.bark();
     secondDog.bark();
  }
}
```

The values you send to the constructor are called ACTUAL PARAMETERS (or ARGUMENTS)

What is the output of this program?

2.2: Creating and Storing Objects

Overloading Methods

You can have two or more methods (including Constructors) with the same name as long as they have different Formal Parameters.

This practice is called **OVERLOADING**.

2.2: Creating and Storing Objects

Overloaded Constructors 1/2

```
class Dog {
  int age;
  String name;

  public Dog(String n) {
    age = 0;
    name = n;
  }

  public Dog(int a, String n) {
    age = a;
    name = n;
  }
}
```

Behold, Overloaded Constructors!

2.2: Creating and Storing Objects

Overloaded Constructors 2/2

The number and type of Actual Parameters you send the Constructor determines which one is used.

```
class Driver {

  public static void main (String[] args) {

     Dog firstDog, secondDog;

     firstDog  = new Dog(“Doggo”);
     secondDog = new Dog(3, “Scotty”);

  }
}
```

So how old are Doggo and Scotty?

2.2: Creating and Storing Objects

Default Constructors

```
class Dog {
  int age;
  String name;

  public Dog() {
    age = 0;
    name = "Spot";
  }

 }
```

Default Constructors have no parameters, so they assign default values to the instance variables.

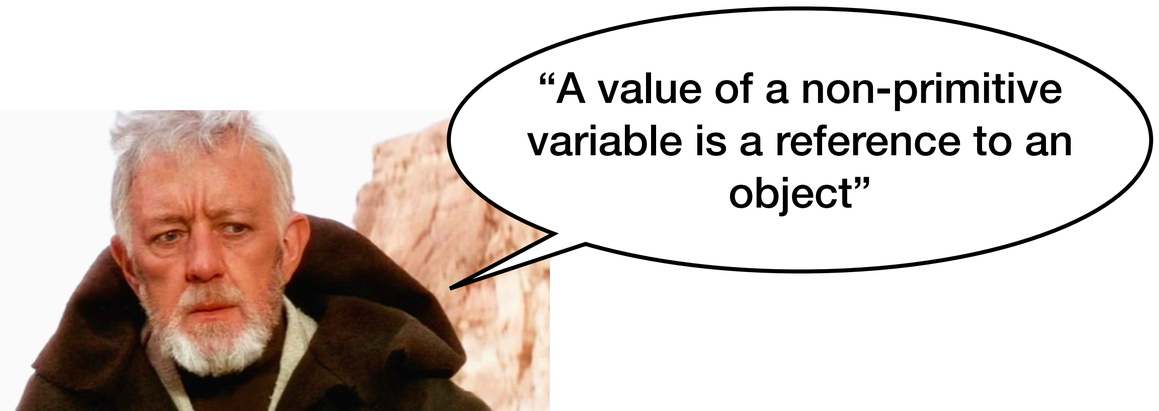
2.2: Creating and Storing Objects

References 1/5

What exactly happens here?

```
Dog thirdDog = new Dog(2, “Snoopy”);
```

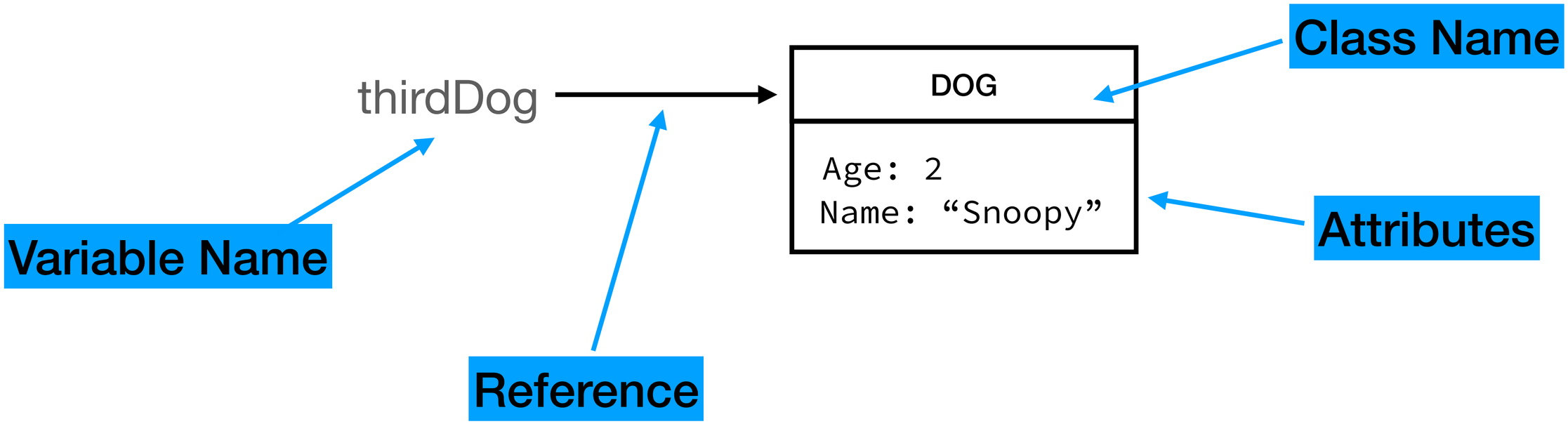
The variable thirdDog is assigned a **reference** to a Dog object. Its value is the reference, not the object itself.



2.2: Creating and Storing Objects

References 2/5

This is often illustrated with this kind of diagram:



2.2: Creating and Storing Objects

References 3/5

In reality, references are not arrows, they are numbers. They represent an address the computer's memory where the object is stored.

2.2: Creating and Storing Objects

References 4/5

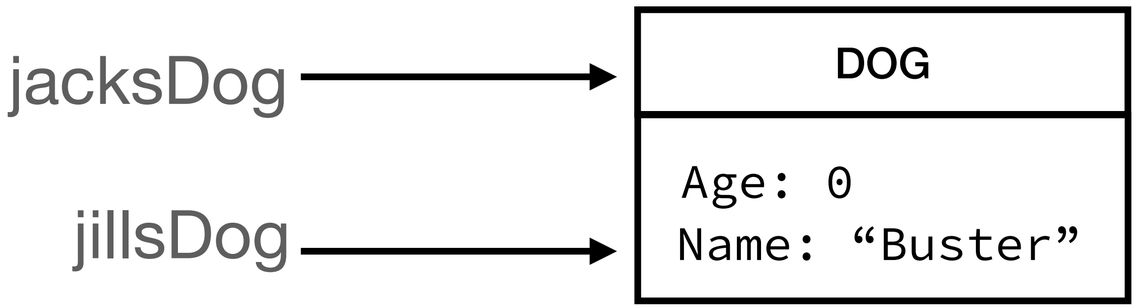
```
class Driver {

  public static void main (String[] args) {

     Dog jacksDog, jillsDog;

     jacksDog  = new Dog(“Buster”);
     jillsDog  = jacksDog;

  }
}
```

What happens here?

2.2: Creating and Storing Objects

References 5/5

The variables hold a reference to the same object!



Surprisingly useful!

2.2: Creating and Storing Objects

Null Pointers 1/2

What about if you don’t assign it a value, like this?

```
Dog hotdog;
```

There is no object, therefore the value of `anotherDog` is **`null`**. This is often symbolized with a downward arrow that points to nothing. Null is not permanent; you can assign it a reference to an object later.

2.2: Creating and Storing Objects

Null Pointers 2/2

Be careful of null pointers! If you try to make this null dog bark:

```
hotdog.bark();
```

Java throws **`NullPointerException`**

Lesson 2.3: Calling a Void Method

Mr. Pelletier, Lord Byng Secondary

2-3: Calling a Void Method

Overview

In this lesson, we will learn what a **void method** is and how to use them!

We will also talk about **Functional Decomposition**.

2-3: Calling a Void Method

What are they?

A void method is a block of code that has a name and can be executed by calling it.

They do not evaluate to anything and therefore you cannot embed them in an expression.

```
System.out.println("Classic example of a void method");
```

```
class ReferenceLetter {

  String teacherName;

  public ReferenceLetter() {
    teacherName = "Leonard Pelletier";
  }

  public void print() {
    printIntro();
    printBody();
    printSignature();
  }

  public void printIntro() {
    System.out.println("To Whom It " +
                       "May Concern:");
    System.out.println("");
  }

  public void printBody() {
      System.out.println( "It has been an honour " +
            "to teach this student. If you don't " +
            "accept them to your university, you " +
            "are a fool!");
      System.out.println("");
    }

    public void printSignature() {
      System.out.println("Sincerely,");
      System.out.println(teacherName);
    }
  }
```

```
class Driver {

 public static void main(String[] args) {
    ReferenceLetter letter = new ReferenceLetter();
    letter.print();
  }
}
```

Lesson 2.4: Calling a Void Method with Parameters

Mr. Pelletier, Lord Byng Secondary

2-4: Calling a Void Method with Parameters

Overview

In this lesson, we will learn how to use parameters to make methods more customizable!

2.4: Calling a Void Methods with Parameters

Example

It would be nice if we could make the reference letters more customized! That's what parameters are for.

```
class ReferenceLetter {
   String teacherName;

   public ReferenceWriter(String n) {
     teacherName = n;
   }

   void print() {
      . . .
   }
}
```

```
class ReferenceLetter {
  String teacherName;

  public ReferenceLetter() {
    teacherName = "Leonard Pelletier";
  }

  public ReferenceLetter(String name) {
    teacherName = name;
  }

  public void print(String name, int years) {
    printIntro();
    printBody(name, years);
    printSignature();
    printFooter();
  }

  public void printIntro() {
    System.out.println("To Whom It " +
                       "May Concern:");
    System.out.println("");
  }

  public void printBody(String name, int years) {
    System.out.println("It has been an honour " +
       "to teach " + name + ". I have taught " +
       "this student for " + years + " years. " +
       "If you don't accept them to your " +
       "university, you are a fool!");
    }

    public void printSignature() {
      System.out.println("");
      System.out.println("Sincerely,");
      System.out.println(teacherName);
    }

    public void printFooter() {
      System.out.println();
    }
  }
```

```
class Main {
  public static void main(String[] args) {
    ReferenceLetter myLetter = new ReferenceLetter();
    myLetter.print("Cal Culator", 1);
    myLetter.print("Al Gebra", 3);

    ReferenceLetter schniederLetter = new
                ReferenceLetter("Katrina Schnieder");
    schniederLetter.print("Bob Loblaw", -1);
  }
}
```

Lesson 2.5: Calling a Non-void Method

Mr. Pelletier, Lord Byng Secondary

2-5: Calling a Non-Void Method

Overview

In this lesson:

- What a non-void method?
- How do you use non-void methods?
- What is the `return` keyword?

2.5 Calling a Non-Void Method

What are they?

Methods are blocks of code with names that can be executed by calling it.

Non-void methods **evaluate** to a value of a specified type and therefore you can embed them in expressions.

You use the `return` keyword to tell Java what the method evaluates to.

2.5 Calling a Non-Void Method

Examples 1/3

Here's the Celsius to Fahrenheit question from Problem Set 1A:

```
public static void main (String[] args) {
    System.out.print(“Enter a temperature in Celcius: ");
    double cTemp = input.nextDouble();
    double fTemp = cTemp * 9 / 5 + 32;
    System.out.println(cTemp + " C = " + fTemp + " F");
}
```

2.5 Calling a Non-Void Method

Examples 2/3

Here's the Celsius to Fahrenheit question from Problem Set 1A:

```
public static void main (String[] args) {
    System.out.print("Enter a temperature in Celcius: ");
    double cTemp = input.nextDouble();
    double fTemp = cTemp * 9 / 5 + 32;
    System.out.println(cTemp + " C = " + fTemp + " F");
}
```

You can put this calculation in a non-void method

2.5 Calling a Non-Void Method

Examples 3/3

```
public void convertCToF() {
    System.out.print(“Enter a temperature in Celcius: ");
    double cTemp = input.nextDouble();
    System.out.println(cTemp + " C = " + CtoF(cTemp) + " F");
}

public double CtoF(double cTemp) {
    return cTemp * 9 / 5 + 32;
}
```

2.5 Calling a Non-Void Method

Issues 1/2

Make sure the value you are returning matches the method's return type.

```
public int tipCalculator(double percent, double cost) {
    return cost * (1 + percent/100);
}
```

2.5 Calling a Non-Void Method

Issues 2/2

Make sure you return a value under Java's definition of all circumstances.

```
public int abs(int x) {
    if (x >= 0) return x;
    if (x < 0) return (-1 * x);
}
```

What's wrong with this?

2.5 Calling a Non-Void Method

Why? 1/4

Code Reusability.

For example, if you needed to convert to Fahrenheit in other places in your program, you can reuse your formula by calling the method elsewhere.

2.5 Calling a Non-Void Method

Why? 2/4

Information Hiding

Recall the ReferenceLetter class. What if we had a non-void method that returned the student's GPA?

```
public void printBody() {
    System.out.println( "It has been an honour to teach this student. " +
          "Their GPA was " + calculateGPA() + ". If you don't accept them to " +
          "your university, you are a fool!");
    System.out.println("");
}
```

We don't need to know how this works!

2.5 Calling a Non-Void Method

Why? 3/4

Modeling Mathematical Functions

If you want to model functions, this is the way to do it. The parameter is your input, the return statement is your output. For example, you can model these types of relationships with methods:

$$f(x) = x^2 \qquad g(x) = 2x+3$$

Evaluate: a) f(3) b) g(5) c) f(g(4)) d) g(f(4))

2.5 Calling a Non-Void Method

Why? 4/4

```
public double f (double x) {
   return x * x;
}
```

```
public double g (double x) {
   return 2 * x + 3;
}
```

```
public void evaluations() {
   System.out.println(“A) “ + f(3) + “ B) “ + g(5) );
   System.out.println(“C) “ + f(g(4)) + “ D) “ + g(f(4)) );
}
```

Lesson 2.6: String Objects: Concatenation, Literals, and More

Mr. Pelletier, Lord Byng Secondary

2.6 String Objects

What are Strings?

The String class is a built-in class in Java. It allows us to build and store ordered sequences of unicode characters - in other words, text.

2.6 String Objects

Instantiating String Objects 1/2

You can create a String by calling the String constructor and sending it a String literal:

```
String myGame = new String(“D&D”);
```
✅

But because Strings are hella common, Java provides a shortcut that resembles how primitive values are stored:

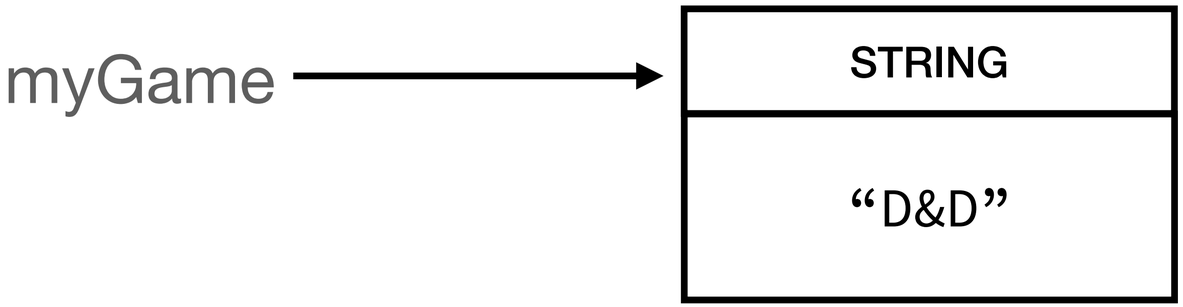
```
String myGame = “D&D”;
```
✅

2.6 String Objects

Instantiating String Objects 2/2

Either way, a new String object is created. The String variable’s value is a reference to that object.

```
String myGame = “D&D”;
```



2.6 String Objects

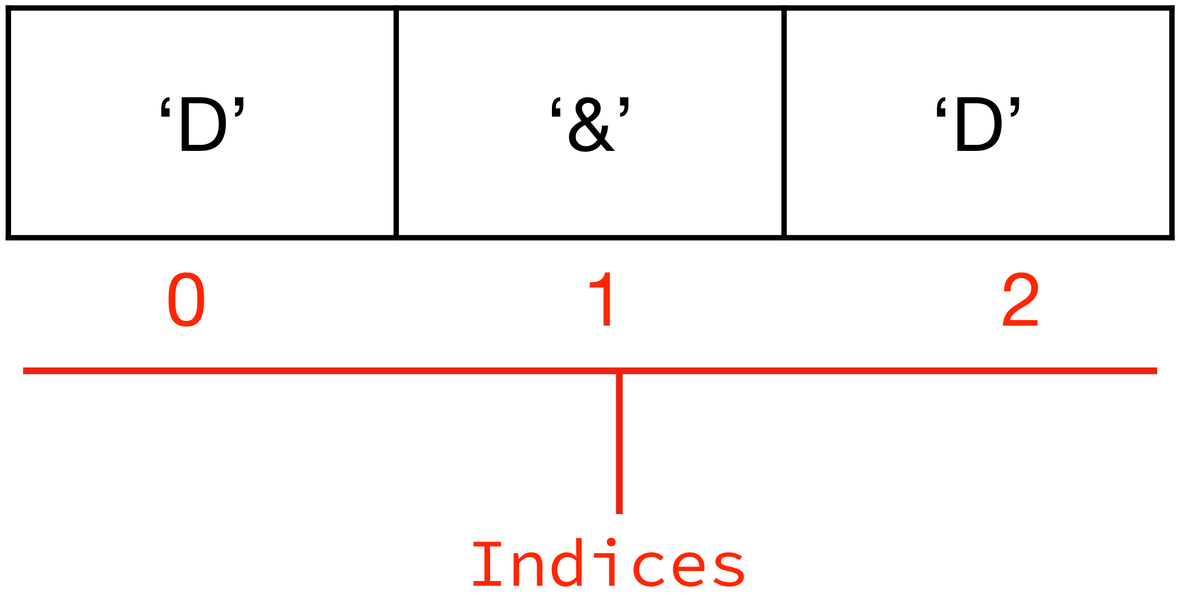
Concatenation

You can combine Strings and other values together using + and +=.

```
int num = 4;
String word1 = “Star”;
String word2 = word1 + “ Wars ” + num;
word2 += “: A New Hope”;
```

2.6 String Objects

Structure of a String



2.6 String Objects

Escape Sequences 1/2

Some characters are hard to represent in code, so Java uses **escape sequences** to code them.

Character	Description	Escape Sequence
“	Double quote mark	\“
	Newline	\n
\	Backslash	\\

2.6 String Objects

Escape Sequences 2/2

For example, how would you represent this:

```
“c:\doom.exe”
“c:\civ.exe”
```

As a single String literal that could be printed with **one** System.out.print statement?

```
System.out.println(“\“c:\\doom.exe\”\n\“c:\\civ.exe\””);
```

2.6 String Objects

Strings are Immutable 1/5

Strings are immutable - a String object cannot change its value after it is created! But how does that make sense with concatenation? Ex:

```
String myStr = “Hello”;
myStr = myStr + “ World”;
```

So isn’t the value changing? No! Here’s what’s actually happening:

2.6 String Objects

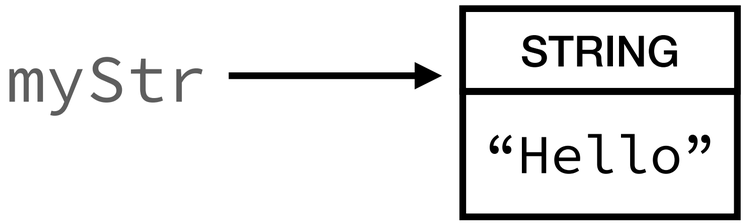
Strings are Immutable 2/5

```
String myStr = "Hello";
```

A String literal is an object

The assignment operator points a non-primitive variable to an object

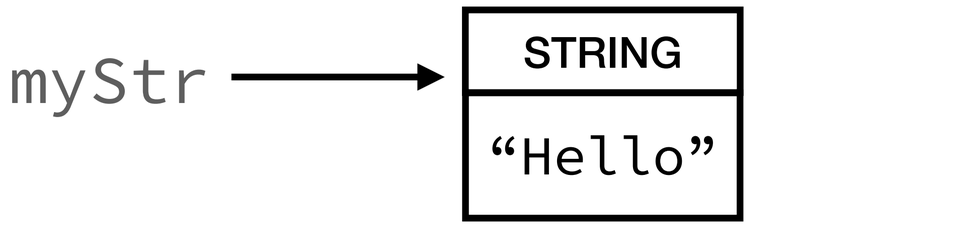
```
String myStr = "Hello";
```



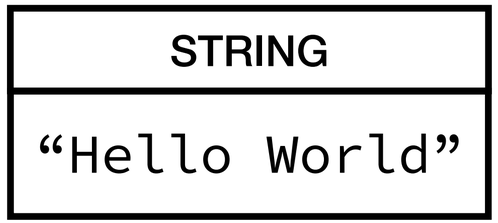
2.6 String Objects

Strings are Immutable 3/5

```
String myStr = “Hello”;
myStr = myStr + “ World”;
```



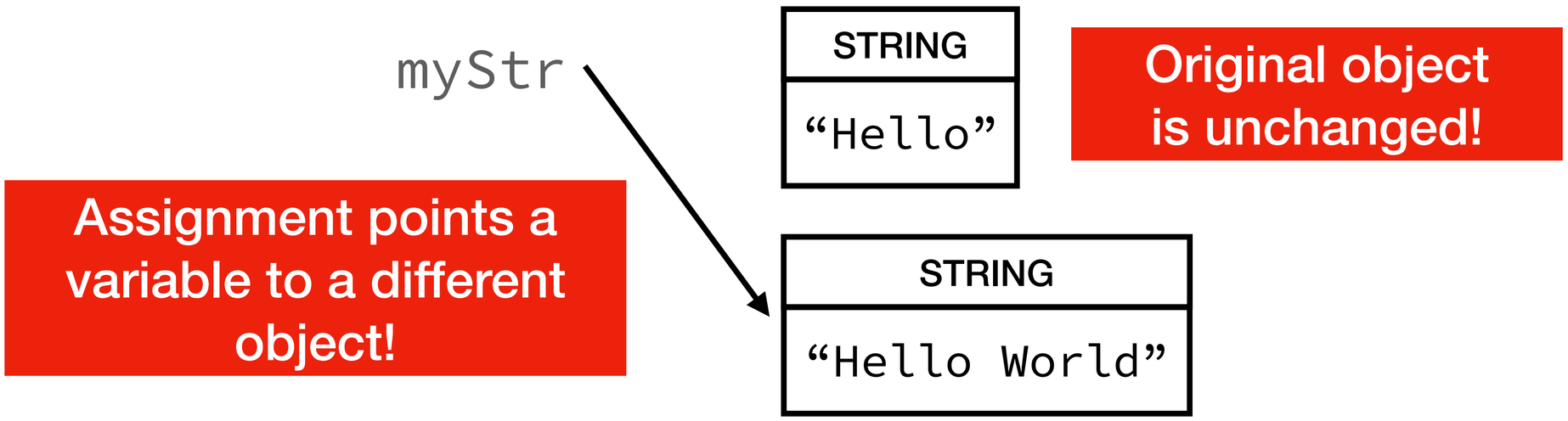
Concatenation produces a NEW object!



2.6 String Objects

Strings are Immutable 4/5

```
String myStr = “Hello”;
myStr = myStr + “ World”;
```

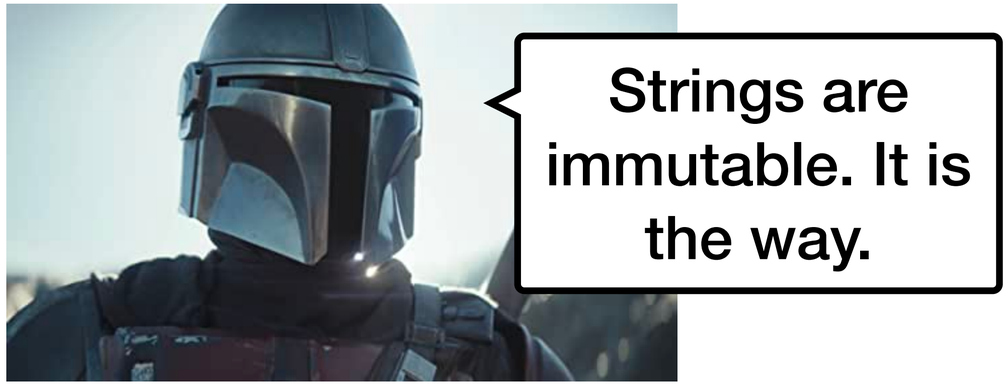


2.6 String Objects

Strings are Immutable 5/5

The Java Virtual Machine is always running in the background and watching out for objects that no longer have references. When it finds one, it frees up that object's memory (essentially, deletes it).

This process is called "Garbage Collection."



Lesson 2.7: String Methods

Mr. Pelletier, Lord Byng Secondary

2.7 String Methods

Overview

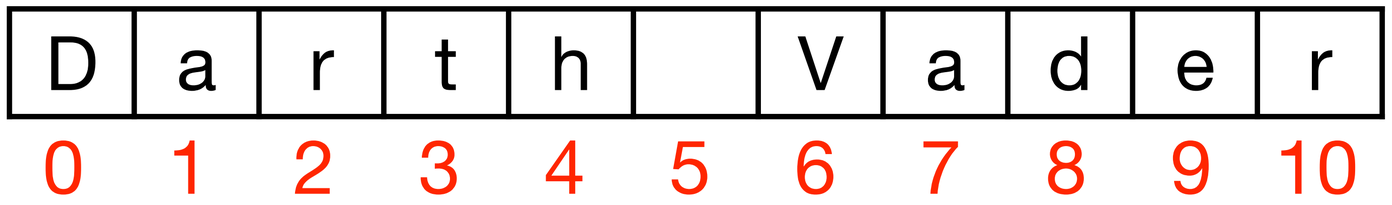
In this lesson, we will see what you can do with Strings. You learn some methods that can be called from Strings and see examples of them in use.

2.7 String Methods

length

`int length()` - returns the number of characters in a String object

```
String str = “Darth Vader”;
```




```
System.out.println(str + “ has “ + str.length() +
“ letters… right?”);
```

2.7 String Methods

indexOf

`int indexOf(String str)` - returns the starting index of the first occurrence of `str` found in a String object searching from left to right, or -1 if there is no occurrence.

Example:

D	a	r	t	h		V	a	d	e	r
0	1	2	3	4	5	6	7	8	9	10

```
String str = “Darth Vader”;
System.out.println(str.indexOf(“art”));
System.out.println(str.indexOf(“good”));
```

2.7 String Methods

equals 1/2

`boolean equals(String other)` - returns true if this String is exactly the same as `other`. Example:

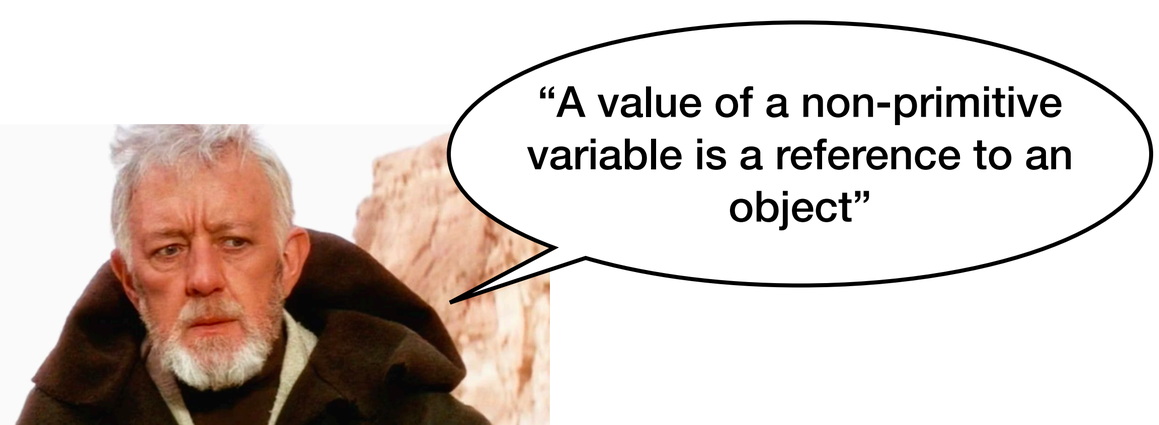
```
String str1 = "Darth Vader";
String str2 = “Anakin Skywalker”;
System.out.println(str1.equals(str2)); //false
```

2.7 String Methods

equals 2/2

Can you use == to compare if two strings are equal?

```
String str1 = “Luke”;
String str2 = “Luke”;
System.out.println (str1 == str2);
```



2.7 String Methods

compareTo 1/2

`int compareTo(String other)` - compares whether this string or other comes first in lexicographical order. Returns a negative int if this string comes first, zero if they are equal, or a positive int if other comes first.

2.7 String Methods

compareTo 2/2

```
String as = “Anakin Skywalker”;

String dv = “Darth Vader”;

System.out.println(as.compareTo(dv));    // -int

System.out.println(dv.compareTo(as));    // +int

System.out.println(as.compareTo(as));    // 0
```

2.7 String Methods

Substring 1/2

Substring is an overloaded method! Here are the signatures:

```
String substring (int start, int end)
String substring (int start)
```

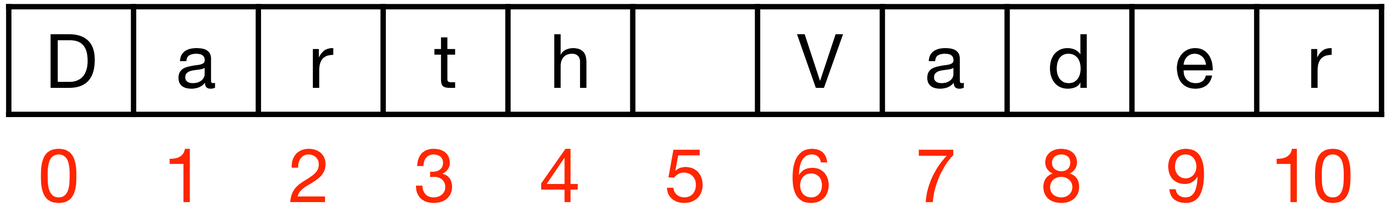
The method returns a part of this String from the start index up to but not including the end index. Note: that start is INCLUSIVE but end is EXCLUSIVE.

If you just provide one actual parameter, it starts at the start index and goes all the way to the end of the String.

2.7 String Methods

Substring 2/2

Examples:




```
String str = “Darth Vader”;
System.out.println(str.substring(1,4)); //art
System.out.println(str.substring(6)); //Vader
```

Lesson 2.8: Wrapper Classes: Integer and Double

Mr. Pelletier, Lord Byng Secondary

2.9 Integer and Double

Integer Class	
`Integer(int value)`	Constructs a new `Integer` object that represents the specified `int` value
`Integer.MIN_VALUE`	The minimum value represented by an `int` or `Integer`
`Integer.MAX_VALUE`	The maximum value represented by an `int` or `Integer`
`int intValue()`	Returns the value of this `Integer` as an `int`
Double Class	
`Double(double value)`	Constructs a new `Double` object that represents the specified `double` value
`double doubleValue()`	Returns the value of this `Double` as a `double`

Lesson 2.9: Using the Math Class

Mr. Pelletier, Lord Byng Secondary

2.9 Using the Math Class

Overview

In this lesson we will learn how to use the Math class to do some fancier calculations.

Math Class	
`static int abs(int x)`	Returns the absolute value of an `int` value
`static double abs(double x)`	Returns the absolute value of a `double` value
`static double pow(double base, double exponent)`	Returns the value of the first parameter raised to the power of the second parameter
`static double sqrt(double x)`	Returns the positive square root of a `double` value
`static double random()`	Returns a `double` value greater than or equal to `0.0` and less than `1.0`

2.9 Using the Math Class

Absolute Value

`static int abs (int x)` - returns the absolute value of x as an int.

`static double abs (double x)` - returns the absolute value of x as a double.

```
int x = Math.abs(-7); // x will be 7
```

2.9 Using the Math Class

Exponents

`static double pow (double base, double exponent)` - returns base to the power of exponent as a double.

```
double x = Math.pow(2,4); // x will be 16.0
```

2.9 Using the Math Class

Square Root

`static double sqrt (double x)` - returns the positive square root of x as a double.

```
double x = Math.sqrt(16); // x will be 4.0
```

2.9 Using the Math Class

Random Numbers

`static double random ()` - returns a random double in the range 0 <= r < 1.

```
int roll = (int) (Math.random()*20) + 1;
// x will a random integer between 1 and 20
```