AP Computer Science A

UNIT 2 TOPIC 1
Intro to Objects Part 2

Do Now

1. Warm Up in Google Classroom!

Do Now!

```
public class CircleRunner {
    public static void main(String[] args) {
        // A. create a Circle object with radius 5.0
        Circle myCircle = new Circle(5.0);

        // B. locate the method defined in the Circle class and write
        //    a line of code to call it on your object
        myCircle.printArea();

        // C. BEFORE running this code, PREDICT what gets printed

        // D. run the code and confirm your prediction! Were you correct?
    }
}
```

Printed output:

```
My area is: 78.5
```

Try This:

- What happens if you were to try and call the method like this:

  ```
  Circle.printArea();
  ```

  What happens and why?

- Are you allowed to name your object "`Circle`" (same name as the class)?

  How about "`circle`" (same name as the class but lowercase)?

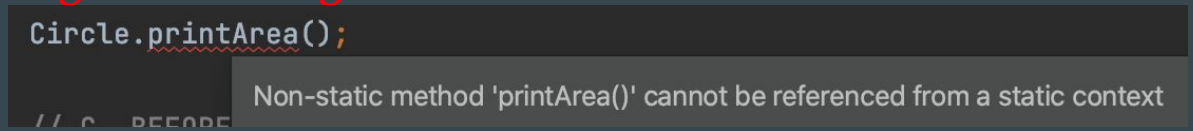
  Explain why or why not.

Try This:

- What happens if you were to try and call the method like this:

  ```
  Circle.printArea();
  ```

  What happens and why? We get this strange error:



  This occurs because we must call the method on the *object*, not on the class from which the object is created (we will discuss the meaning of this error message later)

- Are you allowed to name your object "`Circle`" (same name as the class)?

  How about "`circle`" (same name as the class but lowercase)?

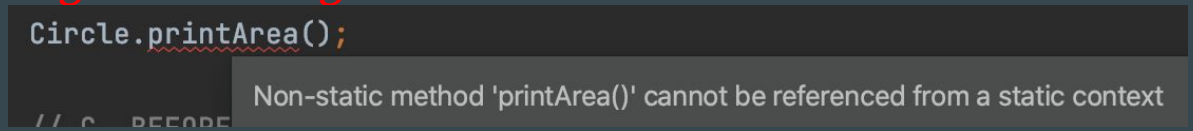
  Explain why or why not.

Try This:

- What happens if you were to try and call the method like this:

  ```
  Circle.printArea();
  ```

  What happens and why? We get this strange error:



  This occurs because we must call the method on the *object*, not on the class from which the object is created (we will discuss the meaning of this error message later)

- Are you allowed to name your object "`Circle`" (same name as the class)? **No!**

  How about "`circle`" (same name as the class but lowercase)? **Yes!**

  Explain why or why not. **Capitalization matters; `Circle` and `circle` are different**

UNIT 2 Using Objects

College Board Standards

Unit 2 Topic 1

MOD-1

2.1 Objects: Instances of Classes

5.A Describe the behavior of a given segment of program code.

ENDURING UNDERSTANDING

MOD-1

Some objects or concepts are so frequently represented that programmers can draw upon existing code that has already been tested, enabling them to write solutions more quickly and with a greater degree of confidence.

LEARNING OBJECTIVE

MOD-1.B

Explain the relationship between a class and an object.

ESSENTIAL KNOWLEDGE

MOD-1.B.1

An object is a specific instance of a class with defined attributes.

MOD-1.B.2

A class is the formal implementation, or blueprint, of the attributes and behaviors of an object.

UNIT 2 Using Objects

College Board Standards

Unit 2 Topic 2

MOD-1 VAR-1

2.2 Creating and Storing Objects (Instantiation)

1.C Determine code that would be used to interact with completed program code.

3.A Write program code to create objects of a class and call methods.

ENDURING UNDERSTANDING

MOD-1

Some objects or concepts are so frequently represented that programmers can draw upon existing code that has already been tested, enabling them to write solutions more quickly and with a greater degree of confidence.

LEARNING OBJECTIVE

MOD-1.C

Identify, using its signature, the correct constructor being called.

ESSENTIAL KNOWLEDGE

MOD-1.C.1

A signature consists of the constructor name and the parameter list.

MOD-1.C.2

The parameter list, in the header of a constructor, lists the types of the values that are passed and their variable names. These are often referred to as formal parameters.

MOD-1.C.3

A parameter is a value that is passed into a constructor. These are often referred to as actual parameters.

MOD-1.C.4

Constructors are said to be overloaded when there are multiple constructors with the same name but a different signature.

MOD-1.C.5

The actual parameters passed to a constructor must be compatible with the types identified in the formal parameter list.

MOD-1.C.6

Parameters are passed using call by value. Call by value initializes the formal parameters with copies of the actual parameters.

LEARNING OBJECTIVE

MOD-1.D

For creating objects:

a. Create objects by calling constructors without parameters.

b. Create objects by calling constructors with parameters.

ESSENTIAL KNOWLEDGE

MOD-1.D.1

Every object is created using the keyword `new` followed by a call to one of the class's constructors.

MOD-1.D.2

A class contains constructors that are invoked to create objects. They have the same name as the class.

MOD-1.D.3

Existing classes and class libraries can be utilized as appropriate to create objects.

MOD-1.D.4

Parameters allow values to be passed to the constructor to establish the initial state of the object.

ENDURING UNDERSTANDING

VAR-1

To find specific solutions to generalizable problems, programmers include variables in their code so that the same algorithm runs using different input values.

LEARNING OBJECTIVE

VAR-1.D

Define variables of the correct types to represent reference data.

ESSENTIAL KNOWLEDGE

VAR-1.D.1

The keyword `null` is a special value used to indicate that a reference is not associated with any object.

VAR-1.D.2

The memory associated with a variable of a reference type holds an object reference value or, if there is no object, `null`. This value is the memory address of the referenced object.

Agenda

- Warm Up!
- Demo: return types and parameters
- U2T1 Lab 2
- Work on project!

return types

- *All* methods have a **return type**
- **`void`** means the method *returns no value*, so a "**void method**" ***has a return type of void*** (note: a void method is still said to have a "void return type"; it is *incorrect* to say that a void method has *no* return type)

example of a "void method" in the `Circle` class with **`void`** return type:

```
// method to print area
public void printArea() {
    double area = 3.14 * radius * radius;
    System.out.println("My area is: " +  area);
}
```

return types

- A method can return a value of any type; if it does return a value, a "`return`" statement **must** appear in the method
- As soon a "`return`" statement is reached, the method returns the indicated value and the method is *immediately* exited (no more code in the method is executed)

example of a method in the `Circle` class with **`double`** return type:

```
// method that returns the area
public double calculateArea() {
    double area = 3.14 * radius * radius;
    return area;
}
```

there **must** be a `return` statement in a method with non-void return type; the value being returned **must** match the return type (in this example, a double)

Calling a void method vs. using the returned value of a non-void method

- **Void methods** just get called; there is no value returned that we need to deal with:

example of calling a **void method** on a `Circle` object within a `CircleRunner` client:

```
Circle myCircle = new Circle(5.0);
myCircle.printArea();
```

- ***Non*-void methods** "hand back" (**return**) a value to the client code that called it; we typically store the returned value in a variable to be used later:

example of calling a ***non*-void** method on a `Circle` object within a `CircleRunner` client;
we store the returned value in a variable and can use the returned value somewhere else:

```
Circle myCircle = new Circle(5.0);
double circleArea = myCircle.calculateArea();
System.out.println("The calculated area is " + circleArea);
```

Calling a void method vs. using the returned value of a non-void method

- We aren't required to store the return value of a ***non***-void method in a variable first, and can instead use the method call's return value "in line":

example of calling a ***non*-void** method and using the return value "in line":

```
Circle myCircle = new Circle(5.0);
System.out.println("The calculated area is " + myCircle.calculateArea());
```

- Note that we **can't** make an "in line" method call with **void** methods, since they don't return a value that can be used in line!

example of *trying* to call a **void** method "in line" -- it fails!

```
Circle myCircle = new Circle(5.0);
System.out.println("The calculated area is " + myCircle.printArea());
```

Operator '+' cannot be applied to 'java.lang.String', 'void'

Methods are their own unique "scopes"

- The code in each method is separate in "scope" from all other methods; Java treats the code in each method entirely separately in terms of variable declaration, naming, and use

example of two different methods in the `Circle` class, each declaring a variable named `area`; you can declare the same variable twice because each declaration is in its own method scope and is treated separately

```java
// method to print area
public void printArea() {
    double area = 3.14 * radius * radius;
    System.out.println("My area is: " +  area);
}

// method that returns the area
public double calculateArea() {
    double area = 3.14 * radius * radius;
    return area;
}
```

returning a variable vs. a calculated value

- A method can return the value of a variable or the result of a direct calculation (IntelliJ prefers the latter):

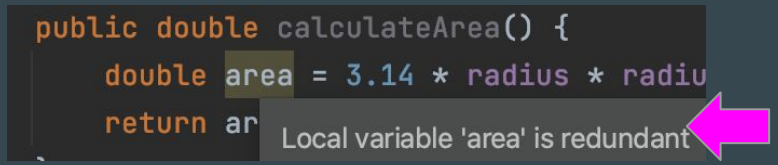
calculating a value and storing it as a variable, then returning the value of the variable:

```
// method that returns the area
public double calculateArea() {
    double area = 3.14 * radius * radius;
    return area;
}
```

OR

returning a calculated value directly, without first storing it as a variable:

```
// method that returns the area
public double calculateArea() {
    return 3.14 * radius * radius;
}
```



IntelliJ says this is unnecessary by giving you a "redundant variable warning", but you can do it this way if you want!

Either of these is OK!
Do whichever you prefer!
You will see both strategies used in practice.

methods can have parameters

- Like constructors, methods can also have parameters:

example of a method in the `Circle` class with a parameter:

```
// method that calculates and returns the volume of a cylinder
// with a circular base with r equal to "radius" and h equal to
// the "height" value passed in as the parameter
public double calculateCylinderVolume(double height) {
    return 3.14 * radius * radius * height; // volume
}
```

- The values get passed in where the method gets called in client code:

example of calling the method with two different parameters in the `CircleRunner` client:

```
Circle myCircle = new Circle(5.0);
double vol1 = myCircle.calculateCylinderVolume(8.5);
System.out.println("Volume 1: " + vol1);
double vol2 = myCircle.calculateCylinderVolume(20.75);
System.out.println("Volume 2: " + vol2);
```

```
Volume 1: 667.25
Volume 2: 1628.875
```

Important! Note that **5.0** is the *radius* used in *both* volume calculations, since `myCircle` was used for both method calls and it was initialized with a radius of 5.0

methods can have parameters

- Like constructors, methods can also have parameters:

example of a method in the `Circle` class with a parameter:

```
// method that calculates and returns the volume of a cylinder
// with a circular base with r equal to "radius" and h equal to
// the "height" value passed in as the parameter
public double calculateCylinderVolume(double height) {
    return 3.14 * radius * radius * height; // volume
}
```

- The values get passed in where the method gets called in client code:

example of calling the method with two different parameters in the `CircleRunner` client:

```
Circle myCircle = new Circle(5.0);
double vol1 = myCircle.calculateCylinderVolume(8.5);
System.out.println("Volume 1: " + vol1);
double vol2 = myCircle.calculateCylinderVolume(20.75);
System.out.println("Volume 2: " + vol2);
```

```
Volume 1: 667.25
Volume 2: 1628.875
```

Important! Note that **5.0** is the *radius* used in *both* volume calculations, since `myCircle` was used for both method calls and it was initialized with a radius of 5.0

methods can have parameters

- Like constructors, methods can also have parameters:

  example of a method in the `Circle` class with a parameter:

```
// method that calculates and returns the volume of a cylinder
// with a circular base with r equal to "radius" and h equal to
// the "height" value passed in as the parameter
public double calculateCylinderVolume(double height) {
    return 3.14 * radius * radius * height; // volume
}
```

- The values get passed in where the method gets called in client code:

  example of calling the method with two different parameters in the `CircleRunner` client:

```
Circle myCircle = new Circle(5.0);
double vol1 = myCircle.calculateCylinderVolume(8.5);
System.out.println("Volume 1: " + vol1);
double vol2 = myCircle.calculateCylinderVolume(20.75);
System.out.println("Volume 2: " + vol2);
```

```
Volume 1: 667.25
Volume 2: 1628.875
```

Important! Note that **5.0** is the *radius* used in *both* volume calculations, since `myCircle` was used for both method calls and it was initialized with a radius of 5.0

methods can have parameters

- Like constructors, methods can also have parameters:

  example of a method in the `Circle` class with a parameter:

```
// method that calculates and returns the volume of a cylinder
// with a circular base with r equal to "radius" and h equal to
// the "height" value passed in as the parameter
public double calculateCylinderVolume(double height) {
    return 3.14 * radius * radius * height; // volume
}
```

- The values get passed in where the method gets called in client code:

  example of calling the method with two different parameters in the `CircleRunner` client:

```
Circle myCircle = new Circle(5.0);
double vol1 = myCircle.calculateCylinderVolume(8.5);
System.out.println("Volume 1: " + vol1);
double vol2 = myCircle.calculateCylinderVolume(20.75);
System.out.println("Volume 2: " + vol2);
```

```
Volume 1: 667.25
Volume 2: 1628.875
```

Important! Note that **5.0** is the *radius* used in *both* volume calculations, since `myCircle` was used for both method calls and it was initialized with a radius of 5.0

calling methods from other methods

- Inside of the Circle class, one method can call another method; since we are *inside* the Circle class itself, it isn't necessary to use dot notation:

example of a method in the `Circle` class calling (and using the return value from) *another* method in the `Circle` class:

```
// method that calculates and returns the volume of a cylinder
// with a circular base with r equal to "radius" and h equal to
// the "height" value passed in as the parameter
public double calculateCylinderVolume(double height) {
    double area = calculateArea();
    return 3.14 * area * height; // volume
}
```

*Note! The strategy above accomplishes the same thing as the strategy below, but the above strategy is **preferred** because it uses logic that already exists rather than rewriting it!*

```
public double calculateCylinderVolume(double height) {
    return 3.14 * radius * radius * height; // volume
}
```

Finding opportunities to **reduce redundancy** is a software engineering best practice!

Before you leave...

- Make sure you have **shared** your project to GitHub (Git → GitHub → Share Project on GitHub)
    - If you made changes since sharing, be sure to Git → Commit then Git → Push
- Log out of your GitHub account on IntelliJ (File → Settings → Version Control → GitHub → click "-" by your name
- Open up a **non**-incognito Chrome window, go to github.com, and make sure you are logged out there
- Close your project in IntelliJ (File → Close Project) and remove it from "Recents" (use the gear icon)