

ACTIVITY 2.1.3

Classes, Subclasses and Testing

INTRODUCTION

Android™ app development takes advantage of **Object Oriented Programming (OOP)** in a wide variety of ways. To this point, you have likely not fully explored many of the features of Object Orientation in Java. **Classes** are a way to group multiple pieces of data and their methods together. Classes allow for effective design of and communication about Android apps.

object-oriented

A type of programming based on objects that represent data.



class

A collection of code that serves a common purpose.

Materials

- Computer with Android™ Studio
- Android™ tablet and USB cable, or a device emulator

RESOURCES

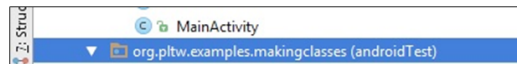
-  **College App Problem Statement**
Resources available online
-  **Free Response Question: Point**
Resources available online

Procedure

- 1 Review the slideshow below.

Unit Testing with JUnit

Create JUnit test classes in the androidTest directory.



Sample Code:

```
public guardianTest extends TestCase {
    guardian p;

    public void setUp() throws Exception {
        super.setUp();
        p = new guardian();
    }

    public void testGetLastName() { assertEquals(p.getLastName(), "Byron"); }

    public void tearDown() throws Exception { super.tearDown(); }
}
```

Computer Science A

© 2016 Project Lead The Way, Inc.

- The example code above will, when run, let you know if `p.getLastName()` is in fact returning "Byron". However, you could add as many test methods as you like and this class will run them all.
- Unit testing is a process by which programmers ensure that individual pieces of a program do what they are supposed to. In Java, it is common to unit test classes. Unit testing is not only performed when code is initially completed, but also whenever it is modified to ensure that the program remains correct.
- <http://codingbat.com/java> is one example of the clarity that unit tests can provide to a programmer. It is also an excellent tool to practice your algorithmic thinking and Java syntax in preparation for the AP exam.
- Occasionally, you will be provided with unit tests that have been pre-written to help you ensure you are doing your work correctly. However, you are strongly encouraged to write your own unit tests to debug and maintain your code throughout this entire course.
- There are many reasons that professional programmers value unit testing. The reasons listed below are adapted from Stack Overflow: <http://stackoverflow.com/questions/67299/is-unit-testing-worth-the-effort>
- Unit tests allow you to make big changes to code quickly. You know your project was working before you made any changes, because you've run the tests. When you make the changes you need to make, you must make sure to get the tests working again. Doing so saves hours because any new errors will be in the newly written code.
- Good unit tests can help document and define what something is supposed to do.
- Unit tests help with code re-use. You can migrate both your code and your tests to your new project, then weak the code until the tests run again.
- The tests and the code work together to achieve better code. Your code could be bad. Your TEST could be bad. But with unit testing, you are banking on the chances of both being bad being low. Often it's the test that needs fixing, but that's still a good outcome.
- When faced with a large and daunting piece of work ahead, writing the tests will get you moving quickly. Unit tests help you really understand the design of the code you are working on. Instead of writing code to do something, you are starting by outlining all the conditions you are subjecting the code to and what outputs you'd expect from that.
- Unit tests give you instant visual feedback; we all like the feeling of all those green lights when we're done. It's very satisfying. It's also much easier to pick up where you left off after an interruption, because you can see how far you got - that next red light that needs fixing.
- Contrary to popular belief, unit testing does not mean writing twice as much code or coding slower. It's faster and more robust than coding without tests, once you've got the hang of it.
- Unit testing helps you realize when to stop coding. Your tests give you confidence that you've done enough for now and can stop tweaking and move on to the next thing.

Overloading

```
public int addNumbers (int op1, int op2, int op3)
```

```
public float addNumbers (int op1, float op2)
```

```
public float addNumbers (float op1, int op2)
```

```
public float addNumbers (int op1, int op2)
```

Computer Science A

© 2016 Project Lead The Way, Inc.

Overloading happens when two methods in the same class have the same identifier but different number, order, or type of parameters.

Overloading and overriding are often confused with each other. Recall from a previous slide that overriding replaces the functionality of a method in the parent class. Overloading, on the other hand, creates additional methods and functionality by defining multiple methods.

Each of the examples above could coexist in the same class, and each could do different things with the parameters it is given, or they could all behave the same. All that matters is that each have a different signature. Identifiers used for the parameters and return types do not qualify as making the method signature “different”. The signature of a method consists of the modifiers, return type, identifier, and list of parameters that precede the block of code.

Here are some examples of method signatures that would cause errors at compile time if they were included in the same class as the ones in the slide. For each of the examples below, which example in the slide does it conflict with?

```
public int addNumbers(int op1, int op2)
```

```
public float addNumbers(int op3, int op4)
```

```
public float addNumbers(int op1, float op2)
```

- 2 Form pairs as directed by your instructor.
- 3 Greet your partner and set team norms for pair programming.

Part I: Create a Class and Its Methods

In the first part of this activity, you will create a new class and some methods for it. You will learn to create a subclass, which is a variation of a class that can hold additional details. Finally, you will learn different ways to test these new elements.

- 4 Open Android Studio. If you have not yet opened a project in Android Studio, please refer to *Activity 1.1.1 Introduction to Android Development*; Part III: Hello Android.

5 Create a new project in Android Studio:

- If you have not yet started a new project in Android Studio, please refer to *Project 1.2.4 Create an Android App*; Part V: Create Something New. In place of MediaLib, name your project **MakingClasses**.
- If you have started a new project before in Android Studio, create a new project with the following values (when no value is specified, leave the defaults):
 - i. Application name: **MakingClasses**
 - ii. Company Domain: **examples.pltw.org**
 - iii. Store your project in your *AndroidProjects* folder.
 - iv. Minimum SDK: **API 22**
 - v. Add an Activity to Mobile: **Empty Activity**
 - vi. Activity Name: **MainActivity**

Once your project is done loading, you'll notice that this project has automatically generated much more code than you may be used to. This code creates the basic elements of an Android app, such as a starting screen.

- 6 Create a new class by selecting **New > Java Class**. Name the class **Arithmetic**. This class will be able to store some numerical data and perform basic arithmetic operations like adding and multiplying. When you create an object from a class as you did in Unit 1, it is also called creating an **instance** of a class. To do this, the class needs a **constructor**.
- 7 Open *Arithmetic.java*, the file that was created when you added the new class, and add lines 3–9 as shown below. The constructor definition is on lines 6–9. It will create an instance of the **Arithmetic** class setting the values of its member variables to some arbitrary values.

```
1: public class Arithmetic {
2:
3:     private int mOperand1;
4:     private int mOperand2;
5:
6:     public Arithmetic(){
7:         mOperand1 = 2;
8:         mOperand2 = 3;
9:     }
10:
11: }
```

- 8 To facilitate debugging, add a `toString()` method to the class following the constructor as shown.

```
1:      mOperand2 = 3;
2:  }
3:
4:  public String toString() {
5:      return "Arithmetic Instance: mOperand1 = " +
        mOperand1 + "; mOperand2 = " + mOperand2 + ".";
6:  }
7:
8: }
```

- 9 In `MainActivity.java`, add the code shown on line 3 and line 5 below.

```
1:  setContentView(R.layout.activity_main);
2:
3:  Arithmetic testArithmetic = new Arithmetic();
4:
5:  System.out.println(testArithmetic);
6: }
```

Line 3 **instantiates** `Arithmetic`, creating a new object of type `Arithmetic`. The variable `testArithmetic` is called an **object reference**, and its data type is the type of object that was used to instantiate it, `Arithmetic`.

- 10 In `Arithmetic.java`, create a method named `add` that adds `mOperand1` and `mOperand2` and returns the result. Lines 4–6 below show the code for this new method.

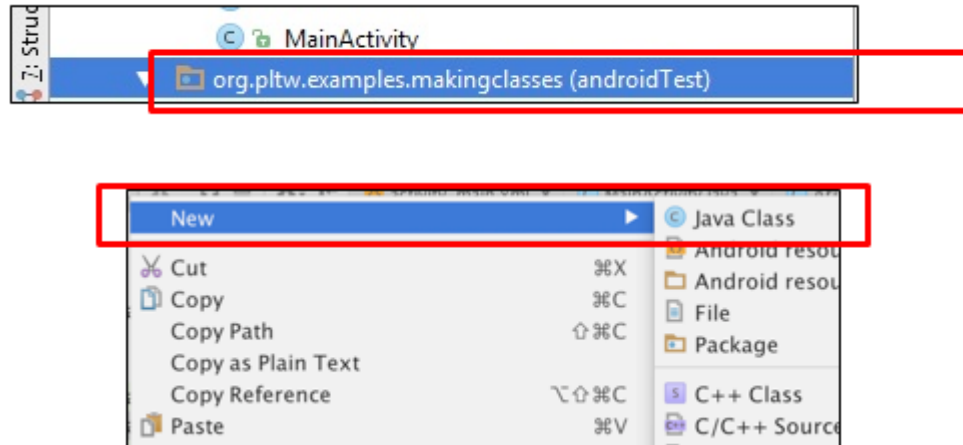
```
1:      return "Arithmetic Instance: mOperand1 = " +
        mOperand1 + "; mOperand2 = " + mOperand2 + ".";
2:  }
3:
4:  public int add() {
5:      return mOperand1 + mOperand2;
6:  }
7:
8: }
```

- 11 Create the `multiply`, `subtract`, and `divide` methods in the `Arithmetic` class on your own.

Part II: Test with JUnit

To test this class and make sure that its methods are behaving correctly, you will use unit testing with a tool called **JUnit**.

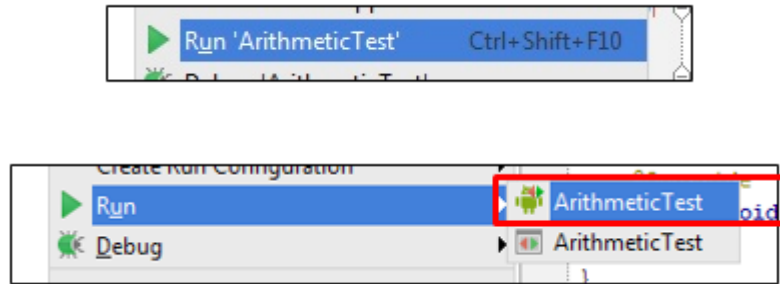
- 12 Set up JUnit. In the project view panel, right-click on the Java package name with “androidTest” following it. Select **New > Java Class**. Name this class `ArithmeticTest`.



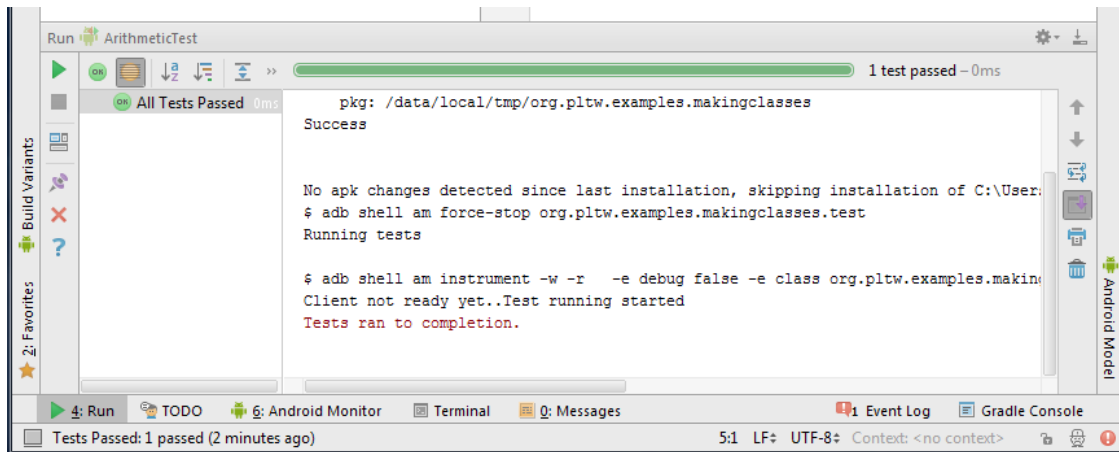
- 13 In the new file named `ArithmeticTest`, insert the following code, making sure to add `extends TestCase` to line 1. You may need to import `junit.framework.TestCase` to get this to work. Android Studio should also prompt you to do this with an **Alt+Enter** option.

```
1: public class ArithmeticTest extends TestCase{
2:
3:     Arithmetic a;
4:
5:     @Override
6:     public void setUp()throws Exception{
7:         super.setUp();
8:         a = new Arithmetic();
9:     }
10:
11:     public void testArithmeticAdd() {
12:         assertEquals(a.add(), 5);
13:     }
14:
15:     @Override
16:     public void tearDown() throws Exception{
17:         super.tearDown();
18:     }
19:
20: }
```

- 14 In the project view panel, right-click **ArithmeticTest** and select **Run > ArithmeticTest** as shown below. The first time through, your options may look like the ones shown in the bottom example.



When your new test class executes, it will run the `setUp` method, and then execute every method whose name begins with `test`, followed by the `tearDown` method. The results of the test run is shown in the Run panel at the bottom of your Android Studio window, similar to the panel below.



Does the execution of `add` pass? How do you know?

- 15 Add test methods for each of the other methods in `Arithmetic` then re-run this test class. Do any of your tests fail, if so, why? Why do you think this kind of testing is useful?

As it exists, `Arithmetic` is not a very flexible class; no instance of `Arithmetic` will be any different from the next.

- 16 Change this by adding a constructor to the `Arithmetic` class that takes two parameters of type `int`: `operand1` and `operand2`. This is called **overloading**, creating two methods with the same identifier but signatures that vary in number, or type, of parameters. In the body of the constructor, assign the value of `operand1` to `mOperand1` and the value of `operand2` to `mOperand2`.
- 17 Acquire `ArithmeticTest2` as directed by your instructor. Run this test to determine if your `Arithmetic` class functions as intended. If it does not, modify it and try again.

What is different in `ArithmeticTest2` than in your `ArithmeticTest`?

overload

Creating one or more methods in the same class with the same identifier BUT different number, type, and/or order of parameters.

Part III: Subclass a Class

Imagine that you wanted to create a class that had all of the functionality of your `Arithmetic` class plus some additional functionality. Think about when you go shopping, and you want to know how much tax you will pay on a purchase. This is a more specific use of the general `Arithmetic` class; you may still want to add and subtract, but you also want to calculate some tax. You *could* add a new method called `calculateTax()` to `Arithmetic`, but that is not good **abstraction**. You would be adding a feature that is only needed occasionally because not all uses of `Arithmetic` would involve shopping or tax.

Rather than clutter `Arithmetic` with features that are not always needed, you can create a **subclass** of the `Arithmetic` class. A subclass has all of the functionality of its **superclass**, in this case `Arithmetic`, plus any other functionality you add to the subclass itself.

- 18 To create a subclass of `Arithmetic`, create a new class in your `MakingClasses` project called `TaxArithmetic`.
- 19 In the new class file named `TaxArithmetic.java`, change the line that defines the class to include `extends Arithmetic` as shown below.

subclass

A class derived or created from another class that is called a superclass or parent class. A subclass inherits or receives all functionality from its superclass.

superclass

The parent, more general class in a class hierarchy.

```
1: public class TaxArithmetic extends Arithmetic {
2: t}
```


The **extends** keyword means that the new class will be a subclass of the stated class and will **inherit** or include its functionality.

- 20 Create a `public` method that returns a `double` called `calculateTax()`. It should have two parameters, a `double` that is the purchase amount and another `double` that is the tax rate.
- 21 In `calculateTax(...)`, implement the algorithm to calculate the tax of a purchase. For example, if you specify a purchase amount of 10.75 and a tax rate of 8.5, the method returns .91375 (ignore the additional decimal places.)
- 22 Test your new subclass in `MainActivity.java`.
 - a. Create an instance of `TaxArithmetic`.
 - b. Call the `calculateTax` method and verify the calculation is correct.
 - c. Demonstrate inheritance; how a subclass inherits the functionality of its superclass (or parent class). To do this, call methods from `Arithmetic` such as `add(...)` or `subtract(...)`, using your `TaxArithmetic` object. For example, you might invoke `ta.add(12, 13)` or `ta.subtract(20, 4)`.
- 23 Overload your `calculateTax` method to calculate the tax based on the addition of the operands in the `Arithmetic` superclass. In place of the purchase amount parameter, add the operands as shown.

NOTE

This will produce an error.

```
1: public double calculateTax(double taxRate) {
2:     int purchaseAmount = mOperand1 + mOperand2
3:     // follow with the rest of your algorithm
4: }
```

What is the error? It is due to how access is granted to a subclass: A subclass inherits all of the data (instance fields) and functionality (methods) of its superclass, but it cannot *directly* access those private fields or methods. Just like other users of a class, a subclass must use public methods to access private instance fields of its superclass.

- 24 Fix the error by using the public methods of the superclass.

extends

When a subclass is created, the `extends` keyword specifies the superclass (or parent class).

inherit or inheritance

When a subclass created automatically gains all the methods and fields of its superclass or parent class.

Part IV: Create a Static Method

Up until this point, you have instantiated objects using blueprints provided by a class. However, instantiating is not always necessary to provide the kind of functionality you have implemented in `Arithmetic`. Static (class) methods are methods that can be called without referencing an instance of a class, such as `Math.random()`. Here, you will create and use your own static method.

- 25 First, define the static method shown below in the `Arithmetic` class.

```
1: public static int add(int operand1, int operand2){
2:     return operand1 + operand2;
3: }
```

- 26 Modify the `testArithmeticAdd` method of `ArithmeticTest2` as shown. Just like `Math.random()` in Lesson 1.1, you called a static method without using an *object*, meaning you don't create an object reference using `new`, you just use the class name. In this case, the class name you use is `Arithmetic`.

```
1: public void testArithmeticAdd() {
2:     System.out.println(Arithmetic.add(2, 2));
3:     System.out.println(Arithmetic.add(2, -5));
4: }
```

- 27 As you may have noticed in your JUnit testing, you can also use an “assertion” to assert, or verify, that the result is what you expected. Change your `testArithmeticAdd` method to verify your static `add` method is working properly.


```
1: public void testArithmeticAdd() {
2:     assertEquals(Arithmetic.add(2, 2), 4);
3:     assertEquals(Arithmetic.add(2, -5), -3);
4: }
```

- 28 Add static methods for your other arithmetic operations in place of your old methods, and modify `ArithmeticTest2` to call the static methods. You may choose to use assertions to verify your code.

When you are done, you should not have instantiated `Arithmetic`, therefore you will **not** have any `Arithmetic` object references anywhere in `ArithmeticTest2`.

Part V: Use a Debugger

In addition to JUnit testing and assertions, you can use a debugger to find errors in a program.

- 29 Android Studio has a built-in debugger. In your browser, navigate to  developer.android.com and search for **debugging with android studio**.
- 30 Read the sections titled “Run your App in Debug Mode” and “Work with Breakpoints” and answer the following questions:
 - a. In your own words, how do you run your app in debug mode?
 - b. What does the Debugger tab show?
 - c. What do breakpoints do when you run your program?
 - d. How do you advance to the next line of code using the debugger?
 - e. How do you switch from debugging back to running your app normally?

- 31 In `ArithmeticTest2`, set a breakpoint on the line that calls `super.setUp()`.
- 32 Run the debugger on `ArithmeticTest2` by right-clicking on the class name in the project view and selecting **Debug > ArithmeticTest2**. Click the **Force Step Into** button. What class does it take you to?

- 33 Rerun the debugger. Experiment with using the Step Over and Step Into buttons until you are able to see the values of `operand1`, `operand2`, `mOperand1`, and `mOperand2` in the Variables panel. What combination of Step Overs and Step Intos did you use to get to that point?

CONCLUSION

1. When is it useful to have multiple instances of a class, rather than creating static methods and fields?
2. Compare and contrast each of the methods you have learned so far for troubleshooting Android apps. They are: logging, unit testing using JUnit, and debugging.