LOYOLA
UNIVERSITY
CHICAGO

COMP 413: Intermediate Object-Oriented Programming

Dr. Robert Yacobellis
Advanced Lecturer
Department of Computer Science

Week 13 Topics

- **Testing in clickcounter and stopwatch**
  - **Group Exercise: create a set of Project 4 unit tests**
  - **Note: save a copy for your team's Project 4 submission!!**
- Event-driven programming
- Possibly time to work on Project 4 in your Groups

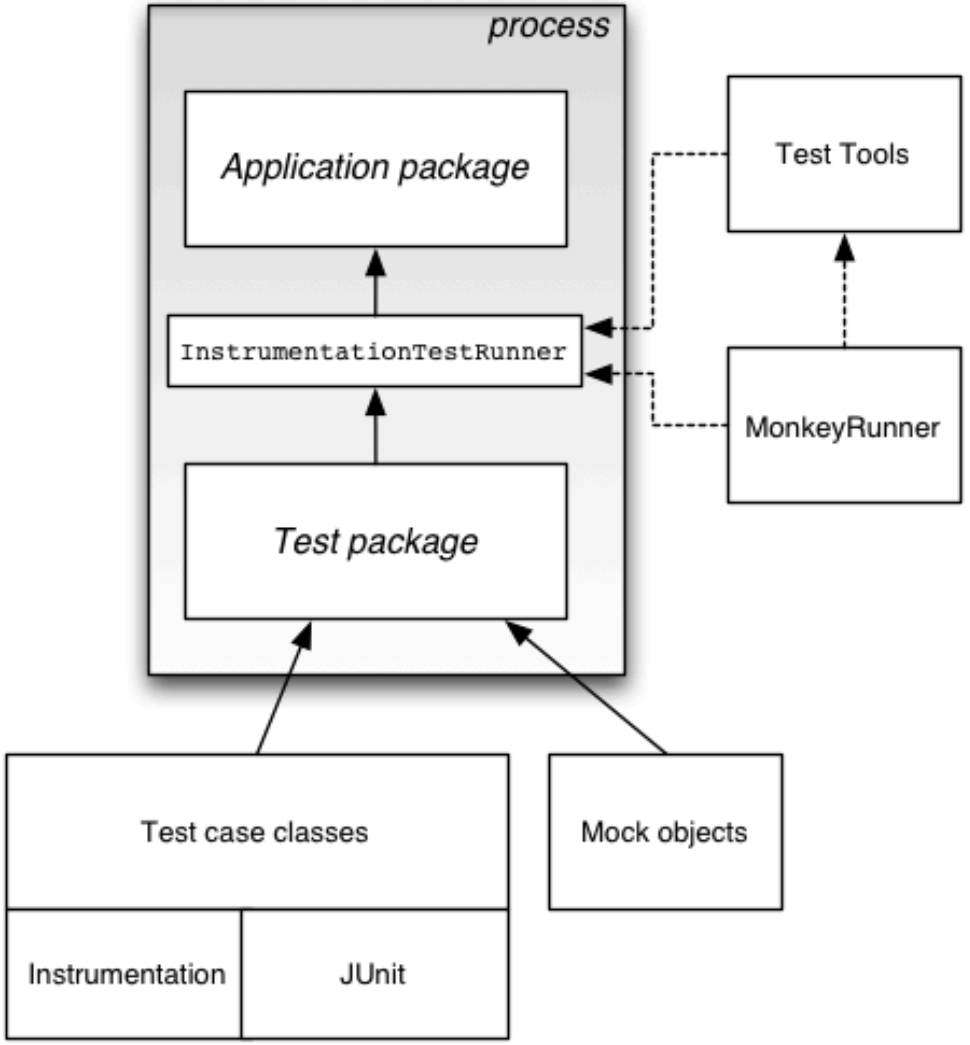
Saving and Restoring Activity State

- There are cases where an Activity is destroyed and restarted, but the user should be unaware of this
  - It is hidden by another (eg, a phone call), then killed to obtain memory
  - The device changes: screen rotations, keyboard or language changes
- In those cases, **onSaveInstanceState()** is called when an Activity is about to be, or may be killed; if that method saves information in the provided Bundle object, when the Activity is restarted via **onCreate()** that Bundle will be passed to it
  - See here for more information about saving information in a Bundle: http://developer.android.com/reference/android/os/Bundle.html
  - Also see http://www.intertech.com/Blog/saving-and-retrieving-android-instance-state-part-1/
- There are other cases where an Activity is destroyed with no intention of restoring it to the same state if it is reactivated
  - Common cases are pressing the Back button, powering down, …

Android Testing Framework

- Key Android testing framework features
  - Android test suites are based on JUnit
    - A JUnit test is a method that tests part of the application
  - Android JUnit extensions provide component-specific test case classes, that include
    - Helper methods for creating mock objects
    - Methods that help control a component's life cycle
  - Test suites are contained in test packages like applications
    - An app called com.mydomain.myapp has a test package name com.mydomain.myapp.test
    - Its tests are run by a test runner, eg, InstrumentationTestRunner
  - Testing tools are available in IDEs and in command-line form
    - There's also an API called monkeyrunner that allows testing devices with Python programs and a command-line tool for stress-testing UIs called the UI/Application Exerciser Monkey

Android Testing Framework



Android Testing Instrumentation

- You can invoke callback methods in test code that allow you to execute an Activity's life cycle step by step
- Testing that an Activity saves and restores its state:

```
// Start the main activity of the application under test
 Activity mActivity = getActivity(); // getActivity() is a key method that's part of the instrumentation API
// in this case it causes onCreate() to be called the first time as the Activity is started

 // Get a handle to the Activity object's main UI widget, a Spinner
 Spinner mSpinner = (Spinner) mActivity.findViewById(com.android.example.spinner.R.id.Spinner01);

 // Set the Spinner to a known position
 mActivity.setSpinnerPosition(TEST_STATE_DESTROY_POSITION);

 // Stop the activity - The onDestroy() method should save the state of the Spinner
 mActivity.finish(); // finish() causes onDestroy() to be invoked

 // Re-start the Activity - the onResume() method should restore the state of the Spinner
 mActivity = getActivity(); // getActivity() will cause onResume() to be invoked

 // Get the Spinner's current position
 int currentPosition = mActivity.getSpinnerPosition();

 // Assert that the current position is the same as the starting position
 assertEquals(TEST_STATE_DESTROY_POSITION, currentPosition);
```

Source: http://www.bogotobogo.com/Android/android12ActivityTestingB.php

Android Activity Testing

- The AndroidTestCase test case base class extends TestCase and Assert and provides all of JUnit's Assert methods plus its setUp() and tearDown() methods
  - It also provides methods for testing Activity permissions
- Component-specific test cases let you manage component life cycle activities; the corresponding API base class is InstrumentationTestCase
  - It provides life cycle control, dependency injection to create mock objects, & user interface interaction (send keystrokes, …)
  - **ActivityInstrumentationTestCase2<T>** is the key derived class, where **T** is the specific Activity class under test
- Now let's look at tests in clickcounter and stopwatch

Android Test Structure

- In Android Studio Projects the "source sets" for source code and test code live in src/main and src/androidTest, respectively
  - Java source code and resources live in java and res sub-directories of these Project directories
    - clickcounter & stopwatch Project tests don't have resources
  - This configuration is managed in the Project's build.gradle file
    - http://tools.android.com/tech-docs/new-build-system/user-guide **has details related to using the Gradle plugin and build.gradle**
- You can run these Android tests on a device or emulator via the Android Debug Bridge (ADB) by right-clicking the module name (eg, ClickCounter or Stopwatch) and selecting Run 'All Tests'
  - In clickcounter/stopwatch, do **View → Tool Windows → Build Variants** and select **Android Instrumentation Tests**; select **Unit Tests** to run those instead
    - **You may have to use gradle testDebug in a Terminal window to make them pass**
  - Note: shapes-android-java (Project 3) did not have Android test code, only Mockito (JVM) test code

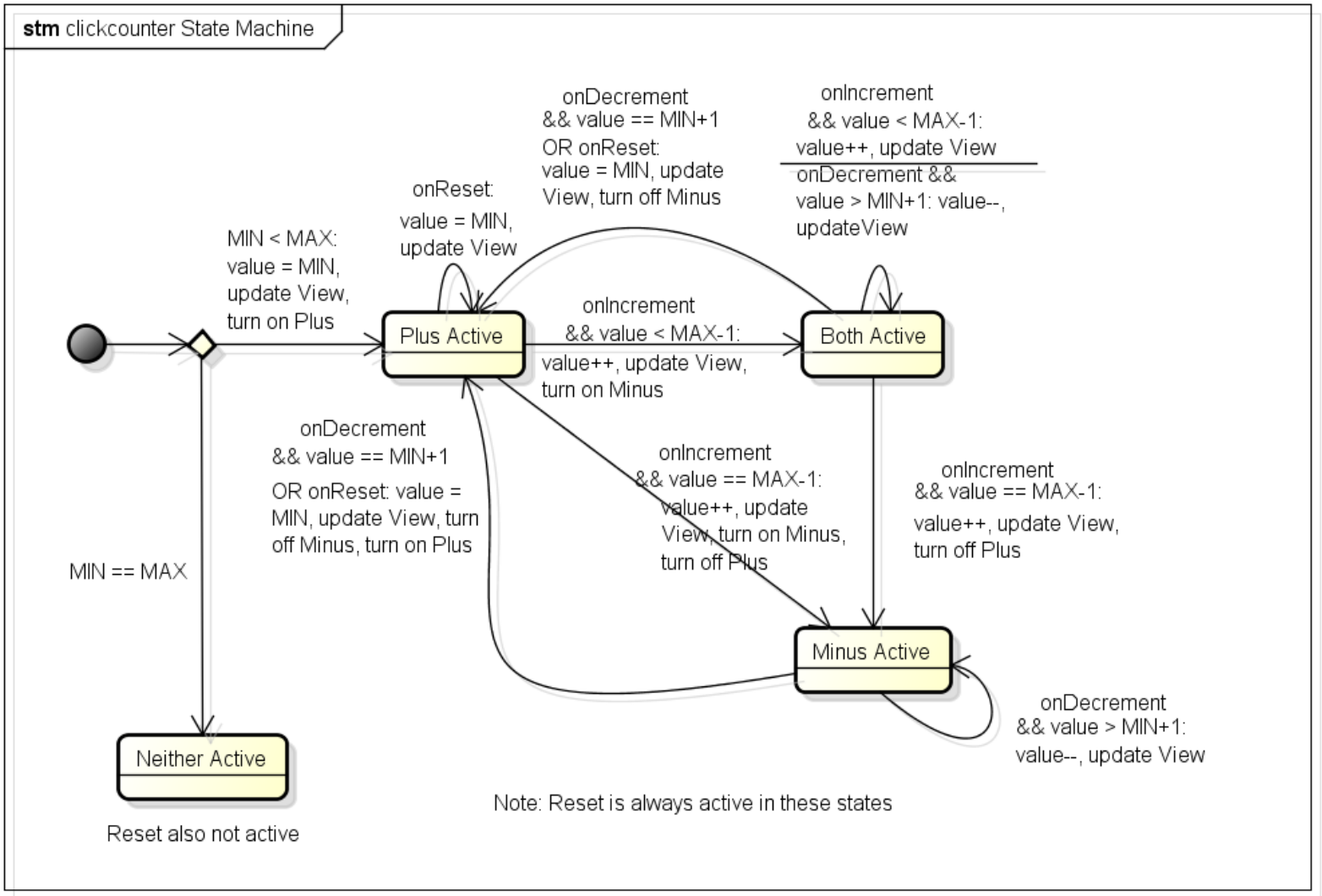
Robolectric Project Test Structure

- Robolectric allows Android Activities to be unit tested in a Java Virtual Machine as opposed to on a device or emulator
- Robolectric dependencies in Android Studio are specified in the build.gradle file, and test source files live in a deep sub-directory of src/test/java (also specified in build.gradle – see URL above)
  - The sub-directory structure matches the package structure in src/main/java (in Projects stopwatch, clickcounter, shapes)
- All directories under src/test/java are visited when you run unit tests via gradle testDebug at the Project level (in Project directory)
  - gradle testDebug runs the <ActivityName>Robolectric.java file; it extends the abstract Activity test in a src/main subdirectory
  - gradle testDebug also finds and runs all of the Test.java files in src/test/java; these files may extend unit test files in main/java

clickcounter Unit Tests – androidTest

- ClickCounterActivityTest.java in the src/androidTest directory delegates to AbstractClickCounterActivityTest.java in src/main; its test methods run in the UI thread on a device or emulator:
  - testActivityTestCaseSetUpProperly – Activity is launched OK
  - testActivityScenarioIncReset performs these checks:
    clicking Reset displays 0, enables Inc, and disables Dec
    clicking Inc displays 1; Inc, Dec, and Reset are all enabled
    clicking Reset again displays 0, Inc is enabled, Dec disabled
  - testActivityScenarioIncUntilFull performs these checks:
    clicking Reset displays 0, enables Inc, and disables Dec
    clicking Inc repeatedly increments the display until full
    when display is full, Inc is disabled, Dec and Reset enabled
  - testActivityScenarioRotation checks that the displayed value remains the same if the device is rotated

Implicit clickcounter State Machine



clickcounter Unit Tests: gradle testDebug

- **gradle testDebug** runs ClickCounterActivityRobolectic.java in src/test; it extends and runs AbstractClickCounterActivityTest.java
  - All tests may pass if you use slide 17's **Build Variants** approach, but some may fail
- **gradle testDebug** also runs tests from SimpleBoundedCounterTest & Java8Test (a simple Java 8 lambda test); both are in test/.../misc.boundedcounter.model
- SimpleBoundedCounterTest:
  - testInitiallyAtMin – the counter is initially at its minimum value
  - testIncrement – one can be added to the min counter value
  - testDecrement – one can be subtracted from the max counter value
  - testFullAtMax – the counter is at its maximum value when full
  - testEmptyAtMin – the counter is at its minimum when emptied
  - testPreconditions* – the initial counter has distinct min & max values
  - testGet* – the counter value is consistent across gets
  - testIsFull* – the counter isn't full if decremented at full
  - testIsEmpty* – the counter isn't empty if incremented at empty

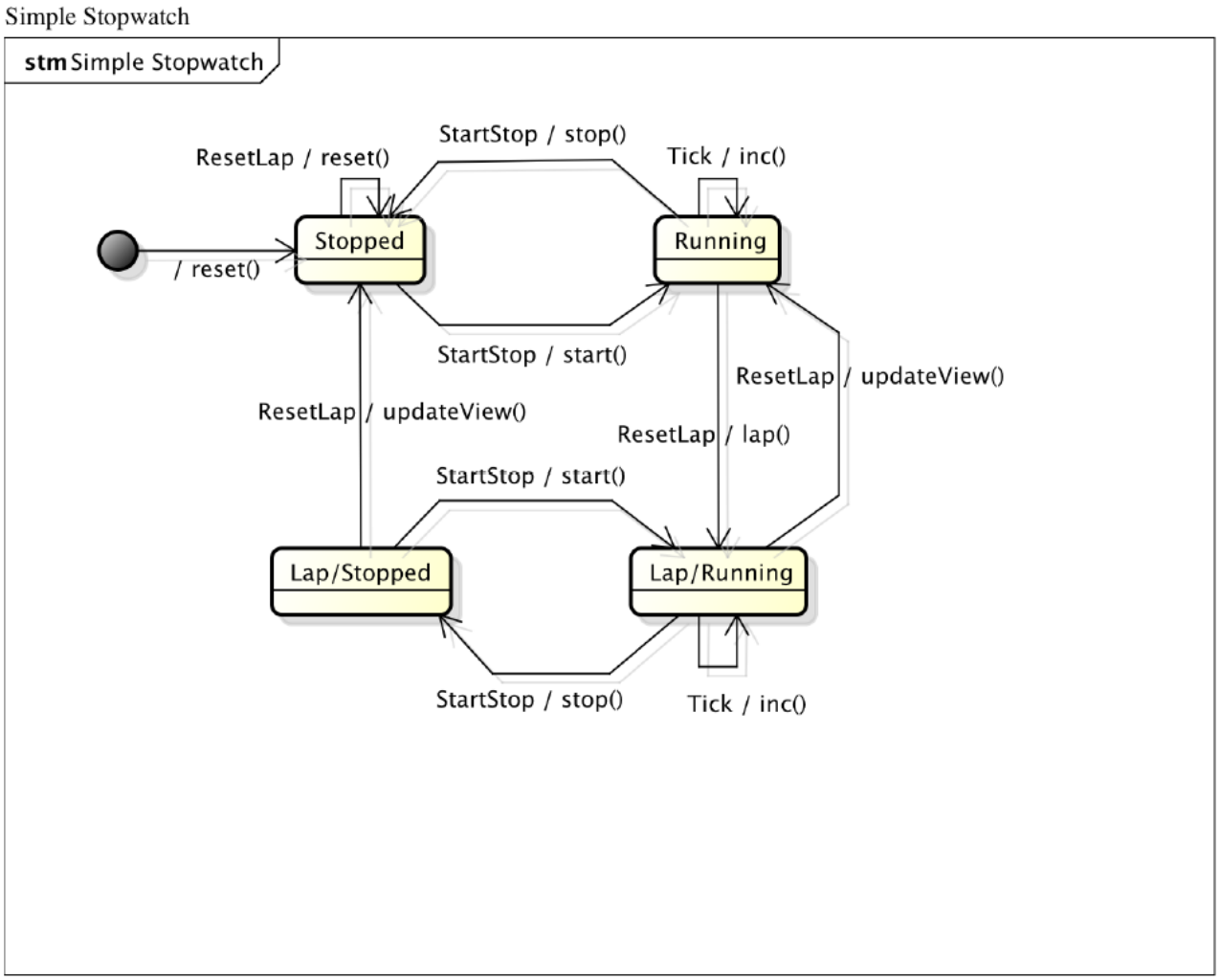
*** inherited from AbstractCounterTest**

stopwatch Unit Tests – androidTest

- StopwatchActivityTest.java in src/androidTest delegates to AbstractStopwatchActivityTest.java in src/main/…/test/android;
  its test methods do not run in the UI thread, but instead use it:
  - testActivityCheckTestCaseSetUpProperly – Activity launched OK
  - testActivityScenarioInit – displayed value is 0
  - testActivityScenarioRun performs these checks:
    the initial displayed value is 0, and Start/Stop can be clicked
    after sleeping 5.5 seconds (not in the UI thread) the displayed value is **5**
    → displayed value checks are run in the UI thread
  - testActivityScenarioRunLapReset performs these checks:
    display is 0, press Start/Stop, wait 5.5 seconds, expect time **5**
    press Reset/Lap, wait **4** seconds, expect time display still 5
    press Start/Stop, expect time display still 5
    press Reset/Lap, expect time display 9 = **5** + **4**
    press Reset/Lap again, expect time display 0

stopwatch-android-java

Reminder: stopwatch State Machine



LOYOLA UNIVERSITY CHICAGO

stopwatch Unit Tests: gradle testDebug 1

- gradle testDebug runs StopwatchActivityRobolectic.java in src/test, which extends & runs AbstractStopwatchActivityTest.java
- Test.java in …/model/clock runs tests in DefaultClockModelTest:
  - Both tests override the onTickListener to be an atomic int
  - testStopped – after sleeping 5.5 seconds, that int is still **0**
  - testRunning – after sleeping 5.5 seconds the int is **5**
- Test.java in …/model/time runs tests in DefaultTimeModelTest:
  - testPreconditions – the model's runtime == **0**, laptime <= **0**
  - testIncrementRuntimeOne – times are correct after one second
  - testIncrementRuntimeMany – times are correct after one hour
  - testLapTime – after 5 ticks the runtime is **+5**, laptime is **+0**; 5 ticks after "clicking" Reset/Lap runtime is **+10**, laptime is **+5**
- Both Default…Tests delegate to corresponding Abstract…Tests

stopwatch Unit Tests: gradle testDebug 2

- Test.java in .../model/state runs tests in DefaultStopwatchStateMachineTest (implemented in AbstractStopwatchStatemachineTest):
  - There's a dependency on an inner mock class that implements TimeModel, ClockModel, and StopwatchUIUpdateListener and replaces their methods with simple integer manipulations, etc.
  - testPreconditions – tests that the initial state is Stopped
  - testScenarioRun – tests that 5 seconds after start the time is **5**
  - testScenarioRunLapReset – verifies the following scenario: time is **0**, press start, state is "Running"; wait 5+ seconds, expect time **5**, press lap, state is "Lap/Stopped"; wait 4 seconds, expect time **5**, press start, state is "Lap/Stopped"; expect time 5, press lap, state is "Stopped", expect time **9**; press lap, state is "Stopped", expect time **0**

Project 4 – Functional Requirements – Slide 1 of 2

- The timer has the following controls:
  - One two-digit display of the form 88.
  - One multi-function button. **(Clicking the button causes a click event.)**
- The timer behaves as follows (part 1 of 2):
  - The timer always displays the remaining time in seconds.
  - Initially, the timer is stopped and the (remaining) time is zero.
  - If the button is pressed when the timer is stopped, the time is incremented by one up to a preset maximum of 99. (The button acts as an increment button.)
  - If the time is greater than zero and three seconds elapse from the most recent time the button was pressed, then the timer beeps once and starts running. **(When the remaining time is greater than 0 the clock model (see stopwatch) is used to send tick events to the state machine.)**

Project 4 – Functional Requirements – Slide 2 of 2

- The timer behaves as follows (part 2 of 2):
  - While running, the timer subtracts one from the time for every second that elapses. **(Caused by a clock model tick event.)**
  - If the timer is running and the button is pressed, the timer stops and the time is reset to zero. (The button acts as a cancel button.)
  - If the timer is running and the time reaches zero by itself (without the button being pressed), then the timer stops and the alarm starts beeping continually and indefinitely.
  - If the alarm is sounding and the button is pressed, the alarm stops sounding; the timer is now stopped and the (remaining) time is zero. (The button acts as a stop button.)
  - The timer handles rotation by continuing in its current state.
- **In your groups, develop a set of Project 4 unit tests, due 11/29 (list only, not tests) → save a copy for your team's submission**

Project 4 Tests to Consider (not complete)

- From clickcounter (button clicking and displayed *timer* value):
  - testActivityTestCaseSetUpProperly: Activity launched OK
  - testActivityScenarioIncUntilFull: clicking *Inc* repeatedly increments the display until full *(the timer will display 99)*
  - testInitiallyAtMin: the *counter value* is initially at its minimum *(0)*
  - test*Increment*: one is added to the (initial) *counter value*
  - Optional: testGet: the *counter value* is consistent across gets
  - Optional: testActivityScenarioRotation: the displayed value *(and state information)* remains the same if the device is rotated *(review how in the clickcounter source code!)*
- From stopwatch (counting *down* the *timer*, i.e. Running):
  - test*Increment*RuntimeOne: *times are* correct after one tick
  - test*LapTime*: after 5 ticks the runtime is *+5 (actually, the displayed timer value should be 5 less than its initial value)*

→ Items in *italics* are differences from existing tests

Project 4 In-Class Exercise – 30 minutes

Name	Group
Killham, Eric John	1
Tapia, Rene	1
Cicale, Julia	1
Rodriguez Orjuela, Jose Luis	1
Mir, Sarfaraz Ali Khan	1
Nowreen, Syeda Tashnuva	1
Goel, Neha	2
Soliz Rodriguez, Percy Gabriel	2
Misra, Anadi	2
Pacheco, Andrea	2
Mehta, Shipra Ashutosh	2
Sindhu, Pinky	2
Al Khofi, Sundas Abdullatif A	3
Liu, Siyuan	3
Meghrajani, Aman Maheshkumar	3
Oakey, Anthony William	3

This Extra Cred ay, November 29

Create a unit t mit it on Sakai

LOYOLA UNIVERSITY CHICAGO

Week 13 Topics

- Testing in clickcounter and stopwatch
  - Group Exercise: create a set of Project 4 unit tests
- **Event-driven programming**
- Possibly time to work on Project 4 in your Groups

Java Inner Classes

From Wikipedia, the free encyclopedia:

In object-oriented programming, **an inner class (aka nested class) is a class declared entirely within the body of another class or interface.** It is distinguished from a subclass.

Inner classes became a feature of the Java programming language starting with version 1.1.

Overview

An instance of a normal or top-level class can exist on its own. By contrast, **an instance of an inner class cannot be instantiated without being bound to a top-level class**.

Let us take the abstract notion of a Car with four wheels. Our wheels have a specific feature that relies on being part of our Car. This notion does not represent the wheels as wheels in a more general form that could be part of any vehicle. Instead it represents them as specific to this particular vehicle. We can model this notion using inner classes as follows:

We have the top-level class Car. Instances of Class Car are composed of four instances of the class Wheel. This particular implementation of Wheel is specific to the car, so the code does not model the general notion of a Wheel which would be better represented as a top-level class. Therefore, it is semantically connected to the class Car and the code of Wheel is in some way coupled to its outer class.

Inner classes provide us with a mechanism to accurately model this connection. We say that our wheel class is **Car.Wheel**, Car being the top-level class and Wheel being the inner class. **Inner classes** therefore **allow for the object orientation of certain parts of the program that would otherwise not be encapsulated into a class.**

Overview of Java Inner Classes

Inner classes, also known as *nested classes,* are classes defined within another class. They may be defined as `public`, `protected`, `private`, or with *package access.* They may only be used "in the context" of the *containing class* (*outer class*, or *enclosing class*), unless they are marked as `static`.

- The outer class can freely instantiate inner class objects within its code; they are automatically associated with the outer class instance that created them. An outer class object can instantiate 0, 1, or many inner class objects.
- Code in some other, non-related class can instantiate an inner class object associated with a specific instance of the outer class if the inner class definition is public (and only if its containing class is public as well).
- If the inner class is `static`, aka a *nested class*, then it can be instantiated without an outer class instance (just using the outer class name, as in **Outer.Inner**).
- An outer class object can see all instance variables and methods in any of its inner class objects, and vice versa for any non-`static` inner class objects

Overview of Java Inner Classes

Inner classes are used to (these uses overlap to some extent):

- create a type of object that is only needed within one class, usually for some short-term purpose (like the **Car.Wheel** example earlier)
- create a utility type of object that cannot be used elsewhere (allowing the programmer to change it without fear of repercussions in other classes)
- create one-of-a-kind interface implementations (such as individualized event handlers – **this is very common, and shows up in Android classes**)
- allow a sort of multiple inheritance, since an inner class can extend a different class than the outer class extends, and an inner class instance has access to both its own private elements as well as the private elements of its outer class object
- implement one-to-many relationships where the classes are tightly coupled (meaning that code for one or both of the classes needs access to many of the private elements of the other class) – the outer class would be the "one" side of the relationship, with the inner class being the "many" side
- provide a specialized form of callback, with which a class may pass very limited access to some of its internal components

- **Collections often implement iterators as inner classes**, enabling them to navigate the structure while not exposing any other aspects of the collection to the outside world.

Types of Java Nested Classes

In Java there are **four types of nested class**:

1. Static member classes, also simply called ***nested classes*** - they are declared **static**. **They do not have an enclosing instance, and cannot access instance variables and methods of the enclosing class**. They are almost identical to non-nested classes except for scope details (they can refer to static variables and methods of the enclosing class without qualifying their names; other classes that are not one of its enclosing classes have to qualify its name with its enclosing class's name). **Nested interfaces are always static.**

2. Inner Classes. **Each instance of these classes has a reference to an *enclosing instance*** (i.e. an instance of the enclosing class), except for local and anonymous classes declared in static context. Hence, **they can implicitly refer to instance variables and methods of the enclosing class**. A reference to the enclosing instance can be explicitly obtained via **EnclosingClassName.this**. There are examples in stopwatch and clickcounter.

Inner classes may not have static variables or methods, except for compile-time constants. When they are created, they must have a reference to an instance of the enclosing class; which means they must either be created within an instance method or constructor of the enclosing class, or (for member and anonymous classes) be created using the special syntax **enclosingInstance.new InnerClass()**.

See next slide for more details of inner classes … there are **3 specific subtypes**

Types of Java Nested Classes

Subclasses of Inner Classes

2.1 Member classes - They are declared outside all methods (hence a "member") and not declared "static".

2.2 Local classes - These are classes which are declared inside the body of a method. They can only be referred to in the rest of the method. They can use local variables and parameters of the method, but only ones which are declared "final". (This is because the local class instance must maintain a separate copy of the variable, as it may out-live the method; so as not to have the confusion of two modifiable variables with the same name in the same scope, the variable is forced to be non-modifiable.)

2.3 Anonymous classes - These are local classes which are automatically declared and instantiated in the middle of an expression. **They can only directly extend one class or implement one interface**. They can specify arguments to the constructor of the superclass, but cannot otherwise have a constructor. **Note that this is the only case where we can write new <interface name>() { … implementation … };!**

→ **In Java 8 lambda expressions can often replace and simplify the use of anonymous classes.**

GUI Callbacks / Event Handling Code

Local inner classes are often used in Java to define callbacks for GUI code. Components can then share an object that implements an event handling interface or extends an abstract adapter class, containing the code to be executed when a given event is triggered.

Anonymous inner classes are also used where the event handling code is only used by one component and therefore does not need a named reference. **These are used in stopwatch and clickcounter.**

This avoids a large monolithic **actionPerformed(ActionEvent)** method with multiple if-else branches to identify the source of the event. This type of code is often considered messy and the inner class variations are considered to be better in all regards.

Java Nested (Inner) Class Example and Inner Classes vs. Java 8 Lambdas

http://examples.javacodegeeks.com/core-java/java-nested-inner-class-example/

http://examples.javacodegeeks.com/java-basics/lambdas/java-8-lambda-expressions-tutorial/ - not on COMP 413 quizzes or tests

http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html - not on COMP 413 quizzes or tests

Threads, Runnables, the Run() Method, …

- Threads are a lightweight way of supporting parallel sets of executing instructions: http://www.slideshare.net/antonkeks/10-threads
  - Note that using asynchronous threads can provoke problems caused by use of mutable data (more later)
- The Runnable interface and run() and start() methods
  - Java multithreading tutorial
  - Java volatile variables, happens-before, and memory consistency
- Stopwatch uses a separate thread to manage its 1-second clock (also useful in Project 4)
  - DefaultClockModel start() method – has an anonymous class!
  - onTickListener
  - ConcreteStopwatchModelFacade
  - DefaultStopwatchStateMachine, StoppedState, RunningState

Week 13 Topics

- Testing in clickcounter and stopwatch
  - Group Exercise: create a set of Project 4 unit tests
- **Event-driven programming**
- Possibly time to work on Project 4 in your Groups

Even More About Java Inner Classes

Inner class code has free access to <u>all</u> elements of the outer class object that contains it, by name (no matter what the access levels of the elements are).

Outer class code has free access to <u>all</u> elements in any of its inner classes, no matter what their access levels.

An inner class compiles to its own class file, separate from that of the outer class (the name of the file will be `OuterClassName$InnerClassName.java`, although **within your code the name of the class will be `OuterClassName.InnerClassName`** – you cannot use the dollar sign version of the name in your code).

An inner class occupies its own memory block, separate from that of the outer class.

The *definition* of an inner class is always available for the outer class to use

- no inner class objects are *automatically* instantiated with an outer class object
- outer class code may instantiate any number of inner class objects – none, one, or many

Syntax:

```
[modifiers] class OuterClassName {
 code
 [modifiers] class InnerClassName {
  code
 }
}
```

Note: inner classes can extend other classes that the outer class cannot!!

Java Inner Class Example

```
public class MyOuter {
 private int x;
 MyOuter(int x, int y) {
  this.x = x;
  new MyInner(y).privateDisplay();
 }
 public class MyInner {
  private int y;
  MyInner(int y) {
   this.y = y;
  }
  private void privateDisplay() {
   System.out.println("privateDisplay x = " + x + " and y = " + y);
  }
  public void publicDisplay() {
   System.out.println("publicDisplay x = " + x + " and y = " + y);
  }
 }
}
```

MyOuter has one property, x; the inner class MyInner has one property, y. The MyOuter constructor accepts two parameters; the first is used to populate x. It creates one MyInner object, whose y property is populated with the second parameter. Note that **the inner class has free access to the private outer class x element, and the outer class has free access to the private inner class privateDisplay() method**. The connection between the two classes is handled automatically.

Referencing Java Inner Classes

If the access for the inner class definition is public (or the element is accessible at package access or protected level to the other class), then other classes can instantiate and reference one or more of these inner class objects. If the inner class is static, then it can exist without an outer class object, otherwise any inner class object you instantiate must belong to an outer class instance

For code that is not in the outer class, a reference to a static or non-static inner class object (eg, the type of a reference variable for it) must use the outer class name, a dot, then the inner class name:

Syntax: *OuterClassName.InnerClassName innerClassVariable*

To create a non-static inner class object from outside the enclosing class, you must attach it to an outer class instance (or the outer class can provide a public factory method to simplify things …).

Syntax: *outerClassVariable*.**new** *InnerClassName*(*arguments*)

For this purpose the *new* operator is a binary operator - it creates a new object of the type on its right belonging to the object on its left. An example:

```
public class Inner2 {
 public static void main(String[] args) {
  MyOuter mo;
  MyOuter.MyInner momi; // reference to an inner class object
  mo = new MyOuter(1, 2);
  momi = mo.new MyInner(102); // creating a non-static inner class object
  momi.publicDisplay();
 }
}
```

To create a static inner class object from outside the enclosing class, you must still reference the outer class name:

Syntax: *OuterClassName*.**new** *InnerClassName*(*arguments*)

Anonymous Class Example, Part 1

```
public class Dog {
   private String breed;  private String name;

   public Dog( String theBreed, String theName ) {
   breed = theBreed; name = theName;
   }

   public String getBreed() { return breed; }
   public String getName() { return name; }

   public int compareTo( Object o ) throws ClassCastException {
   Dog other = (Dog) o;
   int retVal = breed.compareTo( other.getBreed() );
   if ( retVal == 0 )
       retVal = name.compareTo( other.getName() );
   return retVal;
   }
} // Dog
```

Anonymous Class Example, Part 2

```
public void PrintDogsByName( List<Dog> dogs ) {
    List<Dog> sorted = dogs;

    Collections.sort( sorted,
                      new Comparator<Dog> () { // interface*
                @Override
                public int compare( Dog d1, Dog d2 ) {

                      return d1.getName().compareTo( d2.getName() );
                      }
                    );

    Iterator i = sorted.iterator();
    while ( i.hasNext() )
        System.out.println( i.next() );
}
// * Comparator is an interface which the anonymous class implements!
```

Another Anonymous Class Example

```
import javax.swing.*;
Import java.awt.event.*;

public class SwingFrame {
   public static void main( String args[] ) {
   JFrame win = new JFrame( "My First GUI Program" );
   win.addWindowListener(
        new WindowAdapter() {
         @Override
         public void windowClosing( WindowEvent e ) {
                System.exit ( 0 );
         }
        }
        );

   win.setSize( 250, 150 );
   win.show();
   }
} // SwingFrame
// here the anonymous class overrides a "listener" method
```