Class Library, Formatting, Wrapper Classes, and JUnit Testing

- Java Class Library (Packages)
- Formatting Output
- Wrapper Classes and Autoboxing
- JUnit Testing
- Reading for this Lecture: L&L, 3.3 – 3.8

Class Libraries

- A *class library* is a collection of classes that we can use when developing programs
- The *Java standard class library* is part of any Java development environment
- Its classes are not part of the Java language per se, but we rely on them heavily
- Various classes we've already used (`System`, `Scanner`, `String`) are part of the Java standard class library (Look them up on Sun website)
- Other class libraries can be obtained through third party vendors, or you can create them yourself

Packages

- The classes of the Java standard class library are organized into *packages*
- Some packages in the standard class library are:

Package	Purpose
java.lang	General support
java.applet	Creating applets for the web
java.awt	Graphics and graphical user interfaces
javax.swing	Additional graphics capabilities
java.net	Network communication
java.util	Utilities
javax.xml.parsers	XML document processing

The import Declaration

- When you want to use a class contained in a package, you can use its *fully qualified name*

```
java.util.Scanner scan = ...
```

- Or you can *import* the package containing the class and just use the class name `Scanner`

```
import java.util.Scanner;

Scanner scan = ...
```

- To import all classes in a particular package, you can use the * wildcard character

```
import java.util.*;
```

The import Declaration

- All classes of the `java.lang` package are imported automatically into all programs
- It's as if all programs contain the following line:

```
import java.lang.*;
```

- That's why we didn't have to import the `System` or `String` classes explicitly in earlier programs
- The `Scanner` class, on the other hand, is part of the `java.util` package, so that class must be imported as part of its package

Formatting Output

- Look at NumberFormat and DecimalFormat classes in the text
- They provide you with ways to output numbers with a predefined precision
- For example:

Printing double value of Pi	3.141592…
Printing only 2 decimal digits	3.14

Leading Blanks for Numbers

- There is no Java library mechanism to put leading blanks on digit strings to achieve right hand alignment of column of numbers
- Need to write nested conditional code:

```
System.out.println( "Number is: " +
        (n<10? "   " + n :
        (n<100? "  " + n :
        (n<1000? " " + n :
         n))));
```

Wrapper Classes

- The `java.lang` package contains a *wrapper class* that corresponds to each primitive type:

Primitive Type	Wrapper Class
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`
`void`	`Void`

Wrapper Classes

- The following declaration creates an `Integer` object which is a reference to an object with the integer value 40

```
Integer age = new Integer(40);
```

- An object of a wrapper class is used in situations where a primitive value will not suffice
- For example, some objects serve as containers of other objects
- Primitive values could not be stored in such containers, but wrapper objects could be

Wrapper Classes

- Wrapper classes may contain static methods that help manage the associated type
  - For example, the `Integer` class contains a method to convert digits stored in a `String` to an `int` value:

```
num = Integer.parseInt(str);
```

- Wrapper classes often contain useful constants
  - For example, the `Integer` class contains `MIN_VALUE` and `MAX_VALUE` for the smallest and largest `int` values

Autoboxing

- *Autoboxing* is the automatic conversion of a primitive value to a corresponding wrapper object:

```
Integer obj;
int num = 42;
obj = num;
```

- The assignment creates the appropriate `Integer` object wrapping a value of 42
- The reverse conversion (called *unboxing*) also occurs automatically as needed

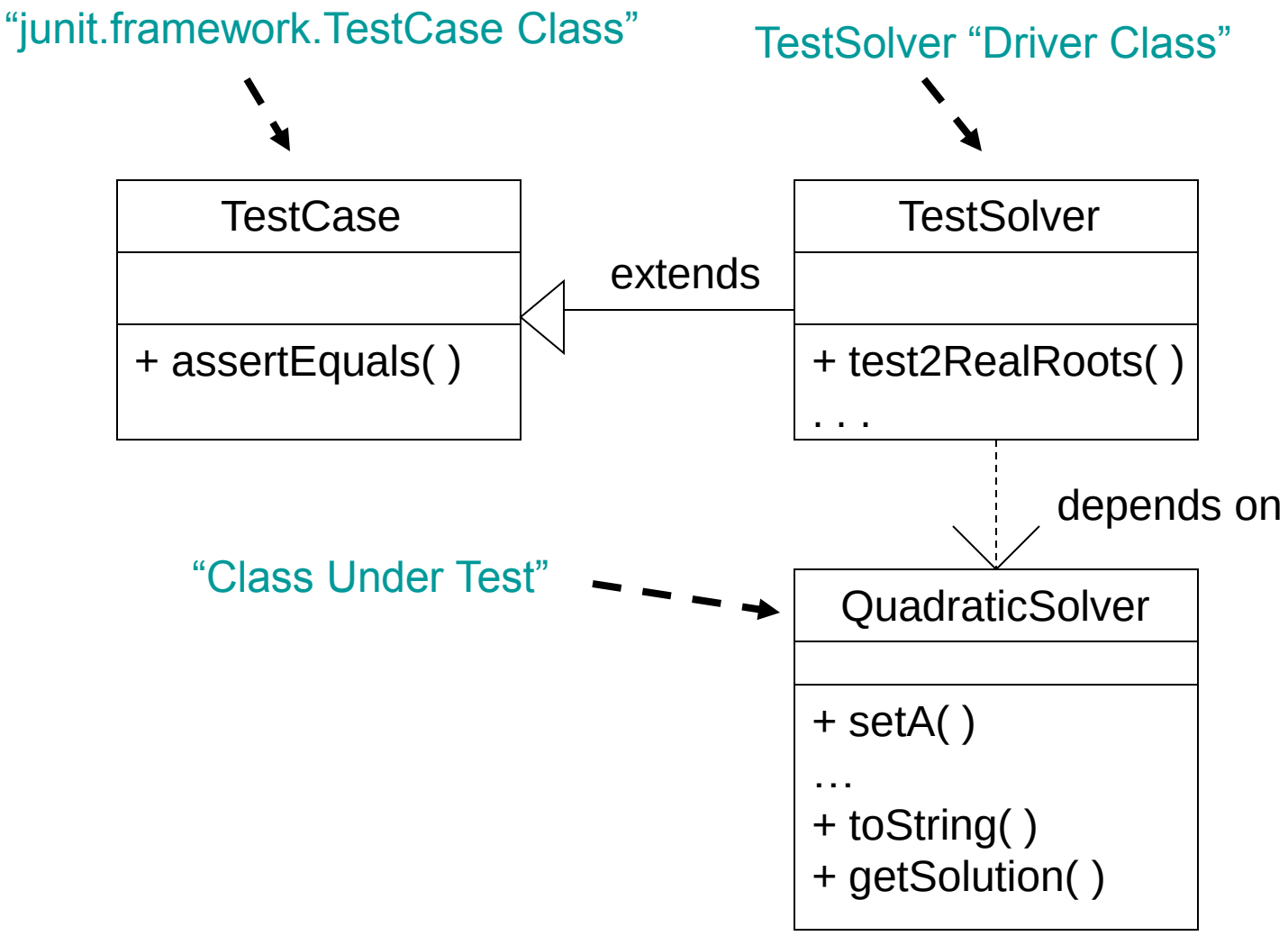
JUnit Testing

- Testing is critical to software quality
- Good test plans are difficult to specify but also difficult to document precisely
- Good testing must be repeatable
- Good testing is tedious
- Testing is a good candidate for automation
- Some methodologies such as “Extreme Programming” mandate daily builds and automated unit testing

JUnit Testing

- In project 1, when we developed our Java code for the QuadraticSolver class, we used the CLI class itself as the “driver” to execute test cases
- We manually entered our test case values and visually verified whether the response provided was correct or not
- This testing process was labor intensive!!
- The JUnit framework helps us build a “test case” class to automate testing of a “class under test”

JUnit Testing



JUnit Testing

- Useful method inherited from TestCase class:

```
assertEquals(Object expected, Object actual)
assertEquals("expected", cut.toString( ));
```

- The assertEquals method flags discrepancies between the “expected” value and the result returned by the “class under test” method( )
- assertEquals method automatically displays the difference between the “expected value” and the actual return value received

JUnit Testing

- Other useful assert… methods

```
assertEquals(double expected_value,
             double actual_value,
             double threshold_value)
```

- Automatically compares absolute difference between first two parameters with a threshold

```
assertEquals(4.3, cut.getDbl(), 0.1);
```

JUnit Testing

- Useful assert… methods for boolean data type

  `assertTrue(boolean actual_value)`
- Automatically expects returned value is true

  `assertTrue(cut.getBoolean());`

  `assertFalse(boolean actual_value)`
- Automatically expects returned value is false

  `assertFalse(cut.getBoolean());`

JUnit Test for QuadraticSolver

```
import junit.framework.TestCase;

public class TestSolver extends TestCase {
   private QuadraticSolver cut;

   public TestSolver()
   {
       // nothing needed here
   }

  // First of six test case methods for the QuadraticSolver class
   public void test2RealRoots()
  {
      assertEquals("Solving: 1x\u00b2 + 0x -1 = 0", QuadraticSolver.getEquation(1, 0, -1));
      assertEquals("Root 1 is 1.0\nRoot 2 is -1.0", QuadraticSolver.getSolution(1, 0, -1));
  }
}
```

JUnit Testing

- Test Case Execution

  1 test failed:

  TestSolver

  test2RealRoots

  test2ImaginaryRoots

  testOnly1Root

  testLinear

  testNoSolution

  testAnySolution

  File: C:\Documents and Settings\bobw\My Documents\bobw\public_html\CS110\Project1\JUnitSolution\TestSolver.java [line: 48]

  Failure: expected:<.....> but was:<...1...>

  (I removed part of "should be" string constant to create error)

JUnit Testing

- The Java code in the TestCase class(es) precisely documents the test cases
- It allows them to be run automatically
- It allows people other than the test designer to run them without knowing the details
- It prevents oversights in identification of any discrepancies in the results