Homework Assignments

- APP B
- Reference on Junit Testing
Classes

- Class Definition
  
  ```java
  class ClassName:
    attributes and methods
  ```

- Public Attribute
  
  ```java
  public class Classname
  {
    attributes and methods
  } // end of class definition
  ```

- Private Attribute
  
  ```java
  private (static) type name (optional = value);
  ```
  Note: Access IS prevented by compiler

- Reserved Word “this”
  
  Used similarly to “self” in Python
  You must use the reserved word “this”.
  Not required in as many places in the code, e.g. not needed in method parameter lists.

- Conventional Word “self”
  
  Used to refer to your own object in Python
  You may use another word, but “self” is the commonly accepted convention.
Classes

• Constructor Method
def __init__ (self, parameter):
    self.parameter = parameter

• Public Method
def name (self, parameters):
    statements

• Private Method
def __name__ (self, parameters):
    statements
Note: A programmer convention only
Access IS NOT prevented by interpreter

• Constructor Method
public ClassName (parameter)
{
    this.parameter = parameter;
} // end of method

• Public Method
def public type name (parameters)
{
    statements;
} // end of method

• Private Method
def private type name (parameters)
{
    statements;
} // end of method
Note: Access IS prevented by compiler
Classes

• Method Return Value
  def name (self, parameters):
    return expression

• Method Overloading
  def name (self, param = None):
    if param is None:
      1st version of statements
    else:
      2nd version of statements

• Method Return Value
  public type name (parameters)
  {
    return expression of type;
  } // end of method

• Method Overloading
  public type name ( ) // no parameter
  {
    1st version of statements;
  } // end of first “name” method

  public type name (type param)
  {
    2nd version of statements;
  } // end of second “name” method
Python “Magic” Methods

• Magic Methods
  __str__(self)            # representation
  __cmp__(self, other)    # compare objects
    (Supports operator overloading for >, <, etc.)
  __add__(self, other)     # and sub, mul, div, etc
    (Supports operator overloading for +, -, *, /, etc )
  __eq__(self, other)       # check equality
  __iter__(self)                # returns an iterator
    (Supports “for item in items” type of loop)
  __del__(self)                # clean up

• Java Equivalents
  public String toString()       // representation
  public int compareTo(that)      // compare objects
    (Supports implementing Comparable interface)
  public boolean equals(that)     // check equality
  public Iterator<T> iterator()  // returns an iterator
    (Supports “for (type item : items)” for-each loop
    and implementing Iterable<T> interface)
  protected void finalize()       // clean up
Creating / Deleting Objects

- **Instantiating an Object**
  
  ```
  myObject = ClassName(. . . )
  # … are values for constructor’s parameters
  ```

- **Instantiating an Object**
  
  ```
  Classname myObject = new ClassName( . . . );
  // … are values for constructor’s parameters
  ```

- **Deleting an Object**
  
  ```
  myObject = None      # deletes object
  # (if there is no alias)
  ```

- **Deleting an Object**
  
  ```
  myObject = null;     // deletes object
  # (if there is no alias)
  ```

- **Creating an Alias**
  
  ```
  yourObject = myObject
  # … both variables refer to the same object
  ```

- **Creating an Alias**
  
  ```
  ClassName yourObject = myObject;
  # … both variables refer to the same object
  ```
Inheritance / Interfaces

- **Inheritance**
  
  # OO Concept: A Cat is an Animal
  
  class Cat(Animal):
    attributes and methods

- **Multiple Inheritance**
  
  class ClassName(Class1, Class2, ...):
    attributes and methods

- **Inheritance**
  
  // OO Concept: A Cat is an Animal
  
  public class Cat extends Animal
  {
    attributes and methods
  } // end of class

- **No Multiple Inheritance**
  
  Java doesn’t support more than one parent class

- **Interfaces**
  
  Java supports implementing multiple interfaces
  
  public class ClassName implements Int1, Int2, ...
  {
  } // end of class
Inheritance / Interfaces

• Polymorphism

class Pet:  # abstract parent class
def makeSound(self):
    raise NameOfError("text")

class Dog(Pet):  # concrete child class
def makeSound(self):
    print "Woof Woof"

class Cat(Pet):  # concrete child class
def makeSound(self):
    print "Meow"

spot = Dog()
spot.makeSound()  # Woof Woof
fluffy = Cat()
fluffy.makeSound()  # Meow

# Attempt to create/use an abstract class
fubar = Pet()
fubar.makeSound()  # raises an Error
    # at run time

• Polymorphism

In Java, a reference to any object may be saved as a reference to the type of a parent class or of any implemented interface:

If Cat class and Dog class extend Pet class, we can do these “widening” conversions:

Dog d = new Dog();
Pet p = d;      // our Pet is a Dog
p = new Cat(); // and is now a Cat
And call any Pet method on variable p:
p.anyPetMethod(); // on Dog/Cat

If a method parameter needs to be a Pet,
public void methodName(Pet p) {...}
we can pass a Dog or a Cat object to it:
methodName(d);  // pass it a Dog
methodName(new Cat()); // or Cat

If Pet is an abstract class, we can’t create a Pet object (causes a compilation error)
Pet p = new Pet();  // compile error
Inheritance / Interfaces

- **Polymorphism**

  If a method definition requires returning a reference to a class or interface, it may return a reference to an object of the class, a child class, or an implementing class.

  If Pet class implements Comparable<T>, Dog and Cat class also implement it. If we invoke a method with a return value of type Comparable<T>:
  ```java
  Comparable<T> c = methodName(...);
  ```

  It can return a Dog or a Cat object:
  ```java
  public Comparable<T> methodName(...) {
      if (some boolean expression) {
          return new Dog();
      } else {
          return new Cat();
      }
  }
  ```
JUnit Testing

- Testing is critical to developing software quality
- Good test plans are difficult to specify but also difficult to document precisely in English
- Good testing must be repeatable
- Good testing is tedious so you will be tempted to cut corners and omit out some needed testing
- Testing is a good candidate for automation
- Some SW development methodologies such as “Extreme Programming” require daily builds and automated testing
JUnit Testing

• In Lab 2, when you develop Java code for the QuadraticSolver class, you use the CLI class itself as the “driver” to enter and execute the test cases from the table in the assignment

• You manually enter the test case values and visually verify whether the response provided is correct or not

• This is labor intensive and error prone!!

• The JUnit framework helps us build a “test case” class to automate testing of a “class under test”
JUnit Testing

“junit.framework.TestCase Class”

TestCase

+ assertEquals()

... 

“Class Under Test”

TestSolver “Driver Class”

TestSolver

+ test2RealRoots()

... 

extends

depends on

QuadraticSolver

+ QuadraticSolver()
+ getEquation()
+ getSolution()
JUnit Testing

• Useful method inherited from TestCase class:
  ```java
  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
- Automatically expects returned value is true
  ```java
  assertTrue(cut.getBoolean());
  ```
- Automatically expects returned value is false
  ```java
  assertFalse(cut.getBoolean());
  ```
import junit.framework.TestCase;

public class TestSolver extends TestCase {
    private QuadraticSolver cut;               // a reference to an object of the “class under test”

    public TestSolver()
    {
        // nothing needed here
    }

    // First of six test case methods for the QuadraticSolver class
    public void test2RealRoots()
    {
        cut = new QuadraticSolver(1, 0, -1);
        assertEquals("Solving: 1x^2 + 0x -1 = 0", uut.getEquation());
        assertEquals("Root 1 is 1.0
Root 2 is -1.0", uut.getSolution());
    }
}
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 an error)
JUnit Testing

- The Java code in the TestCase class(es) precisely documents all 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 of how they work
- It prevents oversights in identification of any discrepancies in the results
- Caveat:
  - When you get errors, check both the test case code and the code of the class under test
  - The error may be caused in either or both places