Reflection basics
Annotations
Unit testing using JUnit 3 and 4

Java Reflection

Java Reflection

- ✓ **Reflection is a feature unique to Java that allows an executing program to examine or “introspect” upon itself, and manipulate properties of the program.**
- ✓ **A Java program can dynamically examine the objects loaded by that application to**
  - ❑ Determine the class of an object.
  - ❑ Create an instance of a class whose name is not known until runtime.
  - ❑ Obtain information about a class' modifiers, fields, methods, constructors, implemented interfaces and superclasses.
  - ❑ Determine constants and method declarations that belong to an interface
  - ❑ Get and set the value of an object's field, even if the field name is unknown to your program until runtime.
  - ❑ Invoke a method on an object, even if the method name and parameters are not known until runtime.
  - ❑ Create a new array, whose size and component type are not known until runtime, and then modify the array's components.
- ✓ **This is accomplished by reflecting on the instance of java.lang.Class that is created by the ClassLoader**

Some of the classes in `java.lang.reflect`

- ✓ AccessibleObject
  - ❑ The AccessibleObject class is the base class for Field, Method and Constructor objects.
  - ❑ Provided to suppress default Java language access control checks
- ✓ Array
  - ❑ The Array class provides static methods to dynamically create and access Java arrays.
- ✓ Field
  - ❑ A Field provides information about, and dynamic access to, a single field of a class or an interface.
- ✓ Constructor<T>
  - ❑ Constructor provides information about, and access to, a single constructor for a class.
- ✓ Method
  - ❑ A Method provides information about, and access to, a single method on a class or interface.
- ✓ Modifier
  - ❑ The Modifier class provides static methods and constants to decode class and member access modifiers.

```
import java.lang.reflect.*;
import java.awt.*;
public class SampleName {
   Button b;
   Class c;

   public SampleName(){
   }

   public static void main
                (String[] args) {
      SampleName sn = new SampleName();
      sn.b= new Button();
      sn.printName(sn.b);
   }
   private void printName(Object o) {
      c = o.getClass();
      String s = c.getName();
      System.out.println(s);
      c = c.getSuperclass();
      s = c.getName();
      System.out.println(s);
   }
}
```

Some of the methods in `java.lang.Class`

- ✓ **Class<?>[] getClasses()**
  - ❑ Returns an array containing Class objects representing all the public classes and interfaces that are members of the class represented by this Class object.
- ✓ **Constructor<?>[] getConstructors()**
  - ❑ Returns an array containing Constructor objects reflecting all the public constructors of the class represented by this Class object.
- ✓ **Field[] getDeclaredFields()**
  - ❑ Returns an array of Field objects reflecting all the fields declared by the class or interface represented by this Class object.
- ✓ **Class<?>[] getInterfaces()**
  - ❑ Determines the interfaces implemented by the class or interface represented by this object.
- ✓ **Method[] getMethods()**
  - ❑ Returns an array containing Method objects reflecting all the public member methods of the class or interface represented by this Class object, including those declared by the class or interface and those inherited from superclasses and superinterfaces.
- ✓ **int getModifiers()**
  - ❑ Returns the Java language modifiers for this class or interface, encoded in an integer.

- ✓ **There are many more methods in java.lang.Class**
  - ❑ Check the API JavaDoc
- ✓ **Note the difference between `getDeclaredXxxx()` and `getXxxx()`**
  - ❑ One gets the inherited items along with all of the items of that kind declared in the class being examined
- ✓ **An instance of `java.lang.Class` is created for every Java class loaded into the JVM**
- ✓ **Note that the return types are from the `java.lang.reflect` package where appropriate**

Drawbacks of Reflection (from the Java Tutorial)

- ✓ **Reflection is powerful, but should not be used indiscriminately. If it is possible to perform an operation without using reflection, then it is preferable to avoid using it. The following concerns should be kept in mind when accessing code via reflection.**
- ✓ **Performance Overhead**
  - ❑ Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations can not be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.
- ✓ **Security Restrictions**
  - ❑ Reflection requires a runtime permission which may not be present when running under a security manager. This is in an important consideration for code which has to run in a restricted security context, such as in an Applet.
- ✓ **Exposure of Internals**
  - ❑ Since reflection allows code to perform operations that would be illegal in non-reflective code, such as accessing private fields and methods, the use of reflection can result in unexpected side-effects, which may render code dysfunctional and may destroy portability. Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.

http://java.sun.com/docs/books/tutorial/reflect/index.html

What is reflection for?

- ✓ **In typical applications you don’t normally need reflection**
  - ❑ As always, there are exceptions
- ✓ **You *do* need reflection if you are working with applications that process other applications such as**
  - ❑ A class browser
  - ❑ A debugger
  - ❑ A GUI builder
  - ❑ An IDE, such as BlueJ or Eclipse
  - ❑ A program to grade student programs
  - ❑ And perhaps the most important use of all – test applications like JUnit

The Class class

- ✓ **To find out about a class, first get its Class object**
  - ❑ If you have an object obj, you can get its class object with
    ```
    Class c = obj.getClass();
    ```
  - ❑ You can get the class object for the superclass of a Class c with
    ```
    Class sup = c.getSuperclass();
    ```
  - ❑ If you know the name of a class (say, Button) at compile time, you can get its class object with
    ```
    Class c = Button.class;
    ```
  - ❑ If you know the name of a class at run time (in a String variable str), you can get its class object with
    ```
    Class c = Class.forName(str);
    ```

Getting the class name

- ✓ If you have a class object c, you can get the name of the class with `c.getName()`
- ✓ `getName` returns the fully qualified name; that is,

  ```
  Class c = Button.class;
  String s = c.getName();
  System.out.println(s);
  ```

  will print

  ```
  java.awt.Button
  ```

- ✓ Class and its methods are in `java.lang`, which is always imported and available

Getting all the superclasses

- ✓ **getSuperclass() returns a Class object (or null if you call it on Object, which has no superclass)**
- ✓ **The following code is from the Sun tutorial:**

```
static void printSuperclasses(Object o) {
   Class subclass = o.getClass();
   Class superclass = subclass.getSuperclass();
   while (superclass != null) {
      String className = superclass.getName();
      System.out.println(className);
      subclass = superclass;
      superclass = subclass.getSuperclass();
    }
}
```

Getting the class modifiers

- ✓ **A Class object has an instance method `getModifiers()` that returns an int**
- ✓ **To "decipher" the int result, we need methods of the Modifier class, which is in `java.lang.reflect`, so:**
  import java.lang.reflect.*;
- ✓ **Now we can do things like:**

```
if (Modifier.isPublic(m))
   System.out.println("public");
```

- ✓ **Modifier contains these methods (among others)**
  - ❑ public static boolean isAbstract(int)
  - ❑ public static boolean isFinal(int)
  - ❑ public static boolean isInterface(int)
  - ❑ public static boolean isPrivate(int)
  - ❑ public static boolean isProtected(int)
  - ❑ public static boolean isPublic(int)
  - ❑ public static String toString(int)
    - ▪ This will return a string such as "public final synchronized strictfp"

Getting interfaces

- ✓ **A class can implement zero or more interfaces**
- ✓ **getInterfaces() returns an array of Class objects**
- ✓ **More code from Sun:**

```
❑ static void printInterfaceNames(Object o) {
    ▪ Class c = o.getClass();
    ▪ Class[] theInterfaces = c.getInterfaces();
    ▪ for (int i = 0; i < theInterfaces.length; i++) {
        – String interfaceName =
                    theInterfaces[i].getName();
        – System.out.println(interfaceName);
❑       }
  }
```

- ✓ **Note that zero-length arrays are perfectly legal in Java**

Examining classes and interfaces

- ✓ **The class Class represents both classes and interfaces**
- ✓ **To determine if a given Class object c is an interface, use c.isInterface()**
- ✓ **To find out more about a class object, use:**
  - ❑ getModifiers()
  - ❑ getFields()   // "fields" == "instance variables"
  - ❑ getConstructors()
  - ❑ getMethods()
  - ❑ isArray()

- ✓ **Each of these is manipulated in the same manner as the Class names and modifiers**
  - ❑ Refer to the API documentation for additional details regarding usage
- ✓ **You can also use reflection to invoke methods**
  - ❑ This uses the .invoke() method in the Method class
  - ❑ The invoke command takes an obj, and an args[]
- ✓ **To change values of data fields in in objects.**
  - ❑ field.setField(obj, 12.34) for example

Constructors

- ✓ **If c is a Constructor object, then**
  - ❑ c.getName() returns the name of the constructor, as a String (this is the same as the name of the class)
  - ❑ c.getDeclaringClass() returns the Class in which this constructor is declared
  - ❑ c.getModifiers() returns the Modifiers of the constructor
  - ❑ c.getParameterTypes() returns an array of Class objects, in declaration order
  - ❑ c.newInstance(Object[] initargs) creates and returns a new instance of class c
    - ▪ Arguments that should be primitives are automatically unwrapped as needed

Manipulating arrays

- ✓ Reflection also allows for the creation and manipulation of arrays.
- ✓ The example to the below is fairly simple. It creates an array of size 5, and then sets the 3rd location of the array to a string. It then gets that string and prints it.

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

public class array1 {
   public static void main(String args[]){
      Class cls = Class.forName("java.lang.String");
      Object arr = Array.newInstance(cls, 5);
      Array.set(arr, 3, “this sets location 3 in the array");
      String s = (String)Array.get(arr, 3);
      System.out.println(s);
}
```

Why Use Reflection

- ✓ **Reflection solves problems within object- oriented design:**
  - ❑ Flexibility
  - ❑ Extensibility
  - ❑ Pluggability
- ✓ **Software becomes “soft”**
  - ❑ Or at least softer
- ✓ **Reflection provides**
  - ❑ High level of object decoupling
  - ❑ Reduced level of maintenance
- ✓ **Reflection solves problems caused by**
  - ❑ The static nature of the class hierarchy
  - ❑ The complexities of strong typing

Design Strategies for Using Reflection

✓ **Challenge switch/case and cascading if statements**

- ❑ Rationale
  - ▪ The switch statement should scream “redesign me” to the developer
  - ▪ In most cases, switch statements are used to perform a pseudo subclass operations
- ❑ Steps
  - ▪ Redesign using an appropriate class decomposition
  - ▪ Eliminate the switch/case statement
  - ▪ Consider a design pattern approach
- ❑ Benefits
  - ▪ High level of object decoupling
  - ▪ Reduced level of maintenance

Design Patterns and Reflection

- ✓ **Many of the object oriented design patterns can benefit from reflection**
  - ❑ Reflection extends the decoupling of objects that design patterns offer
  - ❑ Can significantly simplify design patterns
- ✓ **Factory Method example**
  - ❑ Item 1 in Effective Java is **Consider static factory methods instead of constructors**

Factory Without Reflection

```
public static Shape getFactoryShape (String s) {
   Shape temp = null;
   if (s.equals (“Circle”))
      temp = new Circle ();
   else
      if (s.equals (“Square”))
         temp = new Square ();
   else
      if (s.equals (“Triangle”)
         temp = new Triangle ();
   else
   // …
   // continues for each kind of shape
return temp;
}
```

```
Factory With Reflection
public static Shape getFactoryShape (String s) {
   Shape temp = null;
   try
   {
      temp = (Shape) Class.forName (s).newInstance ();
   }
   catch (Exception e)
   {
      //do something meaningful here
   }
   return temp;
}
```

This method continues to work no matter how many different kinds of Shapes are added to the application

In summary

- ✓ **Java reflection is useful because it allows one to retrieve information about classes, and data structures by name within an executing Java program.**
- ✓ **It also allows one to manipulate those objects**
- ✓ **This lecture comes mainly from the following sources**
  - ❑ http://java.sun.com/docs/books/tutorial/reflect/index.html
  - ❑ http://java.sun.com/docs/overviews/java/java-overview-1.html
  - ❑ http://developer.java.sun.com/developer/technicalArticles/ALT/Reflection/

Annotations

Another way to dynamically discover information about Java classes

Annotations

- ✓ **Annotations are Java code meta-data**
  - ❑ An annotation is an attribute of a program element.
  - ❑ Unlike javadoc tags all annotations are examined at compile time
  - ❑ Some are retained for examination at load or run time

- ✓ **Complements javadoc tags**
  - ❑ Standard annotations
  - ❑ Custom annotations

- ✓ **Annotations are instances of an Annotation Type**
  - ❑ at their most basic level, annotation types are Java interfaces

- ✓ **Syntactically, the annotation is placed in front of the program element's declaration, similar to `static` or `final` or `protected`, `etc..`**

Annotation Categories

- ✓ **Marker annotations**
  - ❑ Marker annotations are used with annotation types that define no members. They simply provide information contained within the name of the annotation itself. An annotation whose members all have default values can also be used as a marker, since no information must be passed in. The syntax of a marker is simply:
- ✓ **@MarkerAnnotation**

- ✓ **Single-value annotations**
  - ❑ Single-value annotations have just a single member, named value. The format of a single-value annotation is:
- ✓ **@SingleValueAnnotation("some value")**

- ✓ **Full annotations**
  - ❑ A full annotation really isn't a category, as much as it is an annotation type that uses the full range of annotation syntax. Here parentheses follow the annotation name, and all members are assigned values:

```
@Reviews({ // Curly braces indicate an array of values
      @Review(grade=Review.Grade.EXCELLENT, reviewer="df"),
      @Review(grade=Review.Grade.UNSATISFACTORY, reviewer="eg",
              comment="This method needs an @Override annotation")
    })
```

Annotatable Program elements:

- ✓ **package**
- ✓ **class, including**
  - ❑ interface
  - ❑ enum
- ✓ **method**
- ✓ **field**
- ✓ **only at compile time**
  - ❑ local variable
  - ❑ formal parameter

Legal Annotation Elements

- ✓ **These elements can be used in an annotation declaration**
  - ❑ primitive types
  - ❑ Strings
  - ❑ enum types
  - ❑ Classes (not general Objects)
  - ❑ arrays of the above
  - ❑ combinations of the above
- ✓ **Declarations use a special form of the interface declaration**

```
@interface Name {
    String first();
    String last();
}
```

Things You Should Know

- ✓ **Annotations go in their own files, just like classes**
- ✓ **Annotation interfaces may not extend anything**
  - ❑ They automatically extend **`java.lang.annotation.Annotation`**.
- ✓ **Their methods take no arguments.**
  - ❑ A field with the same name is automatically declared.
- ✓ **An annotation instance consists of**
  - ❑ the "@" sign
  - ❑ the annotation name
  - ❑ a parenthesized list of name-value pairs.
- ✓ **If only one value named value,**
  - ❑ The name may be omitted.
- ✓ **If no values,**
  - ❑ No parentheses needed.

Three Standard Annotation Types

- ✓ **The pre-defined annotations are**
- ✓ **Override**
  - ❑ **`java.lang.Override`** is used to indicate that a method overrides a method in its superclass.
- ✓ **Deprecated**
  - ❑ **`java.lang.Deprecated`** indicates that use of a method or element type is discouraged.
- ✓ **SuppressWarnings**
  - ❑ **`java.lang.SuppressWarnings`** turns off compiler warnings for classes, methods, or field and variable initializers..

Other Predefined Annotation Types

- ✓ **Four meta-annotations**
  - ❑ **Target**
    - ▪ **This meta-annotation specifies which program elements can have annotations of the defined type.**
  - ❑ **Retention (**SOURCE, CLASS, RUNTIME (CLASS is default))
    - ▪ **This meta-annotation indicates whether an annotation is tossed out by the compiler, or retained in the compiled class file. In cases where the annotation is retained, it also specifies whether it is read by the Java virtual machine at class load.**
  - ❑ **Documented**
    - ▪ **This meta-annotation indicates that the defined annotation should be considered as part of the public API of the annotated program element.**
  - ❑ **Inherited**
    - ▪ **This meta-annotation is intended for use on annotation types that are targeted at classes, indicating that the annotated type is an inherited one.**
- ✓ **Annotations meta-annotated with Retention(RUNTIME) can be accessed via reflection mechanisms.**

Examples

- ✓ **An example with 3 Annotations**
  - ❑ **`Retention`**
    - A meta-annotation
    - how long the element holds onto its annotation
  - ❑ **`Illustrate`**
    - A custom annotation
    - what features of Java 5 a class illustrates
  - ❑ **`Override`**
    - a standard marker annotation for methods that claim to override an inherited method

Things to Note in Example

- ✓ **The following example shows a program that pokes at classes to see "if they illustrate anything."**
- ✓ **Annotations have been annotated.**
  - ❑ These are known as *meta annotations*.
- ✓ **An annotation may be annotated with itself**

Class Annotation Example

```
@Retention(value=RetentionPolicy.RUNTIME)
@Illustrate( {Illustrate.Feature.annotation,
              Illustrate.Feature.enumeration } )
public @interface Illustrate {
    enum Feature {
        annotation, enumeration, forLoop,
        generics, autoboxing, varargs;

       @Override
        public String toString() {
           return "the " + name() + " feature";
        }
    };
    Feature[] value() default {Feature.annotation};
}
```

```
import java.lang.annotation.Annotation;

@Author(@Name(first="James",last="Heliotis"))
@Illustrate(
        {Illustrate.Feature.enumeration,Illustrate.Feature.forLoop})
public class Suggester {
    @SuppressWarnings({"unchecked"}) // not yet supported
    public static void main( String[] args ) {
        try {
           java.util.Scanner userInput =
                                new java.util.Scanner( System.in );
           System.out.print( "In what class are you interested? " );
           Class theClass = Class.forName( userInput.next() );
           Illustrate ill =
              (Illustrate)theClass.getAnnotation( Illustrate.class );
```

… continued …

```
            if ( ill != null ) {
                System.out.println( "Look at this class if you'd " +
                                    " like to see examples of" );
                for ( Illustrate.Feature f : ill.value() ) {
                    System.out.println( "\t" + f );
                }
            }
            else {
                System.out.println(
                          "That class will teach you nothing." );
            }
        }
        catch( ClassNotFoundException cnfe ) {
            System.err.println( "I could not find a class named \"" +
                                cnfe.getMessage() + "\"." );
            System.err.println( "Are you sure about that name?" );
        }
    }
}
```

Compilation and Execution

```
$ javac *.java
Note: Suggester.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

$ java Suggester
In what class are you interested? Suggester
Look at this class if you'd like to see examples of
        the enumeration feature
        the forLoop feature

$ java Suggester
In what class are you interested? Illustrate
Look at this class if you'd like to see examples of
        the annotation feature
        the enumeration feature
```

More Execution

```
$ java Suggester
In what class are you interested? Coin
That class will teach you nothing.

$ java Suggester
In what class are you interested? Foo
I could not find a class named "Foo".
Are you sure about that name?
```

Conclusions

- ✓ **Java annotations are a welcome unification and standardization of approaches to annotating code that language designers and implementors have been doing for decades.**
- ✓ **It is not hard to learn.**
- ✓ **Annotated elements are messy.**
- ✓ **Is this an improvement over XML?**

JUnit 3

Unit Testing Using Reflection

Background

- ✓ **A unit test exercises "a unit" of production code in isolation from the full system and checks that the results are as expected.**
- ✓ **JUnit is a regression testing framework written by Erich Gamma and Kent Beck.**
- ✓ **A test framework provides reusable test functionality which:**
  - ❑ Is easier to use (e.g. don't have to write the same code for each class)
  - ❑ Is standardized and reusable
  - ❑ Provides a base for regression tests
- ✓ **"*Never in the field of software development was so much owed by so many to so few lines of code*" Martin Fowler**

Benefits

- ✓ Identify bugs in the code being tested prior to integrating the code into the rest of the system.
- ✓ It is far easier (i.e. quicker and cheaper) to identify and correct problems (and demonstrate that the solution works) in isolation.
- ✓ Rapid proof that changes in one part of the system do not have unintentional repercussions in another.
- ✓ You can easily run your unit test again and again (hence, regression testing)
- ✓ The "framework" facilitates the testing--it might actually run the test automatically whenever you build or deploy your application
- ✓ Code that goes into production with fewer bugs and requires less rework/downtime.

Pragmatic Unit Testing

- ✓ **General Principles:**
  - ❑ Test anything that might break
  - ❑ Test everything that does break
  - ❑ New code is guilty until proven innocent
  - ❑ Write at least as much test code as production code
  - ❑ Run local tests with each compile
  - ❑ Run all tests before check-in to repository
- ✓ **What to Test: Use Your Right-BICEP**
  - ❑ Are the results RIGHT?
  - ❑ Are all the BOUNDARY conditions correct?
  - ❑ Can you check INVERSE relationships?
  - ❑ Can you CROSS-CHECK results using other means?
  - ❑ Can you force ERROR conditions to happen?
  - ❑ Are PERFORMANCE characteristics within bounds?

http://www.pragprog.com/titles/utj/pragmatic-unit-testing-in-java-with-junit

Exploring Boundaries

✓ CORRECT Boundary Conditions

- ❑ Conformance — Does the value conform to an expected format?
- ❑ Ordering — Is the set of values ordered or unordered as appropriate?
- ❑ Range — Is the value within reasonable minimum and maximum values?
- ❑ Reference — Does the code reference anything external that isn't under direct control of the code itself?
- ❑ Existence — Does the value exist? (e.g., is non-null, non-zero, present in a set, etc.)
- ❑ Cardinality — Are there exactly enough values?
- ❑ Time (absolute and relative) — Is everything happening in order? At the right time? In time?

http://www.pragprog.com/titles/utj/pragmatic-unit-testing-in-java-with-junit

Forming Your Tests

- ✓ **Good tests are A TRIP**
  - ❑ Automatic
  - ❑ Thorough
  - ❑ Repeatable
  - ❑ Independent
  - ❑ Professional

- ✓ **Questions to Ask:**
  - ❑ If the code ran correctly, how would I know?
  - ❑ How am I going to test this?
  - ❑ What else can go wrong?
  - ❑ Could this same kind of problem happen anywhere else?

http://www.pragprog.com/titles/utj/pragmatic-unit-testing-in-java-with-junit

Using JUnit

- ✓ **Generate test cases for object class.**
  - ❑ equivalence partitions
  - ❑ Lots of test cases that test essentially the same path through the code are not terribly useful
- ✓ **Write a JUnit test case class.**
  - ❑ a class that extends TestCase
- ✓ **Run JUnit**
  - ❑ java junit.swingui.TestRunner

Simple Class to test.

```
package org.pittjug.user;

import org.pittjug.exceptions.ValidationException;

public class ChangePassword {
    private String newPassword;
    private String confirmNewPassword;
….
  public void validate throws ValidationException{
      if(!newPassword.equalsIgnoreCase(confirmNewPassword)){
         throw new ValidationException("The new Password and
                                  the Confirm Password must match!");
  }
```

- ✓ **Tests should include**
  - ❑ Passwords match
  - ❑ Passwords do not match
  - ❑ One or more inputs null

A Minimal Test

- ✓ **Must Extend junit.framework.TestCase**
- ✓ **Must have a method named testXXX for each test to be run.**
- ✓ **Use the static methods from junit.framework.Assert to test results.**
  - ❑ SOME of the assert methods

```
static assertEquals(boolean expected, boolean actual)
static assertEquals(double expected, double actual, double delta)
static assertEquals(int expected, int actual)
static assertFalse(boolean condition)
static assertNotNull(java.lang.Object object)
static assertNotSame(java.lang.Object expected, java.lang.Object
actual)
static assertNull(java.lang.Object object)
static assertSame(java.lang.Object expected, java.lang.Object actual)
static assertTrue(boolean condition)
static fail()
```

Simple Test

```
package org.pittjug.user.test;
import junit.framework.TestCase;
…
public class TestChangePassword extends TestCase {

    public void testValidationException() {
        try {
            ChangePassword changer = new ChangePassword();
            changer.setNewPassword("carlhere");
            changer.setConfirmNewPassword("CARLHERE");
            changer.validate();
            fail("Should raise a Validation Exception");
        } catch(ValidationException success) {
            // successful test
        }
    }
}
```

Revised Source

```
package org.pittjug.user;

import org.pittjug.exceptions.ValidationException;

public class ChangePassword {
    private String password;
    private String newPassword;
    private String confirmNewPassword;
….
  public void validate throws ValidationException{
      if(!newPassword.equals(confirmNewPassword)){
         throw new ValidationException("The new Password
and the Confirm Password must match!");
      }
     if(password.equals(newPassword)){
        throw new ValidationException("The new Password
must not match your old Password!");
      }
  ….
```

Added Test

- ✓ **Every time you revise the code, you should revise the tests**
  - ❑ Add more tests
  - ❑ Revise existing tests to reflect code revisions

```
public void testPasswordNotChanged() {
        try {
            ChangePassword changer = new ChangePassword();
            changer.setNewPassword("carlhere");
            changer.setConfirmNewPassword("carlhere");
            changer.setPassword("carlhere");
            changer.validate();
            fail("Should raise a Validation");
        } catch(ValidationException success) {
            // successful test
}
```

So What’s going on?

- ✓ **JUnit 3 uses reflection to find every method that starts with “test”**
  - ❑ creates an instance of your object
  - ❑ executes one method
  - ❑ continues to create new objects for each method
  - ❑ displays summary results of each test

Code example from TestCaseClassLoaderTest.java

```
public void testClassLoading() throws Exception {
        TestCaseClassLoader loader= new TestCaseClassLoader();
        Class loadedClass= loader.loadClass
                                  ("junit.tests.runner.ClassLoaderTest", true);
        Object o= loadedClass.newInstance();
        //
        // Invoke the assertClassLoaders method via reflection.
        // We use reflection since the class is loaded by
        // another class loader and we can't do a successfull
         // downcast to ClassLoaderTestCase.
        //
        Method method= loadedClass.getDeclaredMethod("verify", new Class[0]);
        method.invoke(o, new Class[0]);
}
```

Test Ordering

- ✓ **What Order will the tests be run?**
  - ❑ Alphabetical?
  - ❑ Order present in class?
  - ❑ By hash code value?

- ✓ **It's really up to the jdk and/or the framework!**
- ✓ **So, how do you ensure the order?**

Scenario

- ✓ **Need to test a User Object and Password change.**
  - ❑ Test User Login.
  - ❑ Test Changing the Password.
  - ❑ Test new password saved properly.
- ✓ **Create a test suite**

```
public TestChangePassword(String name) {
    super(name);
}

public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new TestUser("testLogin"));
    suite.addTest(new TestUser(“testChangePassword”);
    suite.addTest(new TestUser(“testLoginNewPassword”);
    return suite;
}
```

setUp() and tearDown()

- ✓ **Allows you to establish conditions necessary to start a test and to return the state of your system after the test is complete.**

- ✓ **setUp() is run before each test!**
- ✓ **tearDown() is run after each test!**

Example setUp

```
private static boolean setUpComplete;

public void setUp(){
  try{
    if(!setUpComplete){
      Class.forName("org.gjt.mm.mysql.Driver");
      Connection conn = DriverManager.getConnection(
              "jdbc:mysql://localhost/jugsoft", "juguser", "catman");
               Statement stmt = conn.createStatement();
               stmt.executeQuery("update usr set password='man' where
                                  loginName='cat'");
      stmt.close();
      conn.close();
      setUpComplete = true;
    }
  }
  catch(ClassNotFoundException cnfe){
       …
```

Some best practices

- ✓ **TestCases (Java classes)**
  - ❑ Usually test a single Java class
  - ❑ Name starts or ends with Test
- ✓ **Tests (Java methods)**
  - ❑ test a single method
    - ▪ testXxx tests method xxx
    - ▪ multiple tests of method xxx are named testXxxYyy, where Yyy describes a particular condition, e.g., testEndsWithEmpty
- ✓ **Label your assertions**
- ✓ **One test per test method**

- ✓ **Many was to run JUnit tests**
  - ❑ Command line
  - ❑ IDE (Eclipse, Netbeans, etc.)
  - ❑ Ant task
- ✓ **Various ways of viewing output (screen, text, XML, HTML, etc.)**

More best practices

- ✓ **Run your tests using a tool: IDE, Ant, Maven**
- ✓ **Tests should be independent**
- ✓ **Same package, different directory**

```
src                         test
  org                         org
    jasig                       jasig
      SomeClass.java              TestSomeClass.java
      OtherClass.java             TestOtherClass.java
```

JUnit 4

Unit Testing With Annotations

“Old” JUnit and JUnit 4.0

- ✓ **We call JUnit 3.8.1 and its predecessors the *“old” JUnit***
- ✓ **In last lecture: studied “old” JUnit**
- ✓ **Now: write a test in old Junit, see same test written in *“new” JUnit 4.0***
- ✓ **Use example that won’t require detailed study of the *software under test***

Testing if a book is in library

- ✓ **All test classes extend from junit.framework.TestCase**
- ✓ **All test methods named with prefix of “test”**
- ✓ **We run code with a *test fixture***
- ✓ **Finally *validate conditions on executed code*, using one of several assert methods.**

```
import junit.framework.TestCase;
public class LibraryTest extends TestCase
{
   public void testBookInLibrary() {
      Library library = new Library();
      boolean result = library.checkByTitle(“King Lear”);
      assertEquals(“Our library should have it!”,
                                    true, result);
   }
}
```

Writing same test in JUnit 4.0

- ✓ You *no longer import* TestCase
- ✓ You do not extend test classes from TestCase;
  - ❑ *test classes are normal class declarations*
- ✓ You do not have to prefix the name of test method with “test”
  - ❑ Uses JSE 5 constructs like static import and annotations

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import junit.framework.JUnit4TestAdapter;
public class LibraryTest // no extension
{
@Test public void bookInLibrary() { // test method
    Library library = new Library();
    boolean result = library.checkByTitle(“King Lear”);
    assertEquals(“Our library should have it!”,
                                   true, result);
   }
  public static junit.framework.Test suite() {
    return new JUnit4TestAdapter(Library.class);
  }
}
```

What you must do in JUnit 4.0

- ✓ **Import Test annotation: org.junit.Test**
- ✓ **Import static org.junit.Assert.assertEquals**
- ✓ **Import JUnit4TestAdapter: junit.framework.JUnit4TestAdapter**
- ✓ **Use @Test annotation to declare method as test method**
- ✓ **Use one of the assert methods for test oracles**
- ✓ **JUnit4TestAdapter required in order to run JUnit 4.0 with old JUnit runner**

Compatibility with old JUnit

- ✓ **Many IDEs don’t yet support Junit 4.0 directly**
- ✓ **JUnit4Adapter ensures that JUnit 4.0 tests can be executed with old runners:**

```
public static junit.framework.Test suite() {
  return new JUnit4TestAdapter(Library.class); }
```

- ✓ **Or use the JUnitCore class in org.junit.runner package through a command line interface:**

```
java org.junit.runner.JUnitCore LibraryTest
```

Setups and teardowns

- ✓ JUnit 4.0 has new annotations for setup and tear down:
- ✓ @Before: any method annotated with @Before executes before every test
- ✓ @After: any method annotated with @After executes after every test
- ✓ Use requires import statements for these keywords

```
import org.junit.After;
import org.junit.Before;
… // more imports
public class LibraryTestUsingSetup {
  private Library library;
  @Before public void runBeforeTest() {
         library = new Library(); }
  @After public void runAfterTest() {
         library = null; }
… }
```

Use of @Before and @After

- ✓ **@Before can be used to express the test fixture as it will be executed before every test**
- ✓ **You can write as many @Before and @After methods as you would like; method names irrelevant; must be public and static**
- ✓ **@Before methods can be inherited: execution is down the inheritance chain**
- ✓ **@After methods can be inherited: execution is up (!) the inheritance chain**
- ✓ **@After methods guaranteed to run even if @Before or @Test methods throw exception**

One-time setup & teardown

- ✓ **JUnit 4.0 has @BeforeClass and @AfterClass annotations for one-time setup/teardown**
- ✓ **similar to TestSetup class in old package junit.extensions**
- ✓ **Requires import of org.junit.Afterclass and org.junit.Beforeclass**
- ✓ **Only one method each for @Beforeclass and @AfterClass allowed within a class (unlike @Before and @After)**
- ✓ **Must be public and static; method name irrelevant; @AfterClass guaranteed to run**

One-time library setup

```
// within class LibraryTest
@BeforeClass public void runOncePrior(){
  library = Library.newInstance(10020);
  library.connectToCentralLibr();
  library.synchrDataWithCentralLibr();
}
// usual test methods and @Before/@After
// methods go here
@AfterClass public void runOnceAfter() {
  library.disconnectFromCentralLibr();
}
```

Expecting exceptions

- ✓ **JUnit 4.0 has easy check for exceptions:**
- ✓ **@Test annotation takes parameter that declares type of Exception that should be thrown In code below, bookNotAvailableInLibrary is test that passes only if NotFoundException is thrown**
  - ❑ Test fails if no exception is thrown
  - ❑ Test also fails if different exception thrown

```
@Test(expected = NotFoundException.class)
public void bookNotAvailableInLibrary() {
   Library library = new Library();
   library.checkByTitle("Some non-existent book");
}
```

Ignoring a test

- ✓ **@Ignore annotation tells runner to ignore test and report that it wasn't run**
- ✓ **Can pass string as parameter to @Ignore explaining why test was ignored, e.g.:**

@Ignore("Database currently down")

Timing out a test

- ✓ **Can pass timeout parameter to test annotation**
- ✓ **Specifies the timeout period in milliseconds**
- ✓ **If test takes more time than speficied, it fails**
- ✓ **E.g. a method annotated with**
- ✓ **@Test (timeout=10)**
- ✓ **fails if it takes more than 10 milliseconds**
- ✓ **Useful for testing quality of service requirements of unit code**

Summary of JUnit 4.0

- ✓ **Requires JDK 5 to run**
- ✓ **Test classes not required to extend from *junit.framework.TestCase***
- ✓ **Test methods don't require prefix "test"**
- ✓ **There is (almost) no difference between the old and new *assert* methods**
- ✓ **Use *@Test* annotations to declare a method as a test case**

- ✓ **Slides marked with the Creative Commons license logo**

creative commons

COMMONS DEED

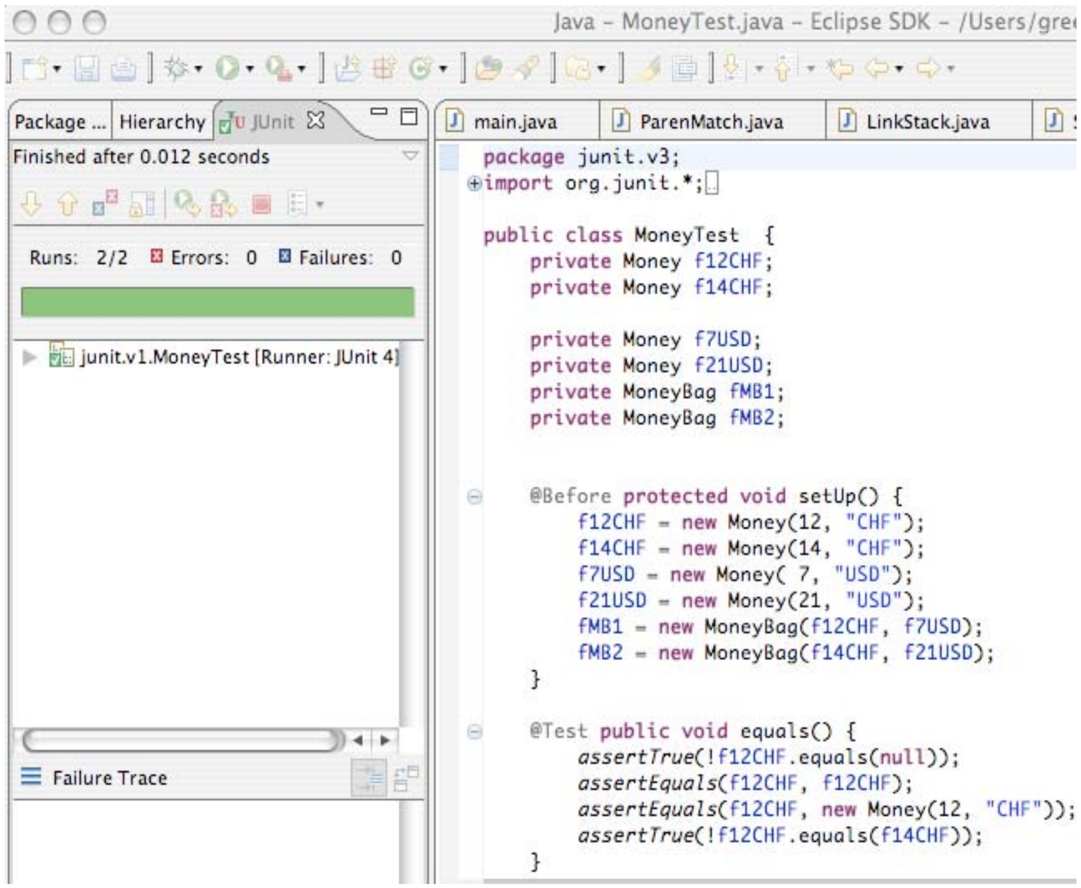
- ✓ **Were originally developed by an unspecified author at the University of Bern and reused under the conditions specified in that license.**

Running tests from eclipse

> Add the junit.jar to the project's buildpath libraries (Project>Properties   )

> Create a JUnit test run configuration for your TestCase

> Open a JUnit view and run the test.



```
Finished after 0.012 seconds
Runs: 2/2   Errors: 0   Failures: 0
junit.v1.MoneyTest [Runner: JUnit 4]
```

```
package junit.v3;
import org.junit.*;

public class MoneyTest  {
    private Money f12CHF;
    private Money f14CHF;

    private Money f7USD;
    private Money f21USD;
    private MoneyBag fMB1;
    private MoneyBag fMB2;


    @Before protected void setUp() {
        f12CHF = new Money(12, "CHF");
        f14CHF = new Money(14, "CHF");
        f7USD = new Money( 7, "USD");
        f21USD = new Money(21, "USD");
        fMB1 = new MoneyBag(f12CHF, f7USD);
        fMB2 = new MoneyBag(f14CHF, f21USD);
    }

    @Test public void equals() {
        assertTrue(!f12CHF.equals(null));
        assertEquals(f12CHF, f12CHF);
        assertEquals(f12CHF, new Money(12, "CHF"));
        assertTrue(!f12CHF.equals(f14CHF));
    }
```

License

✓ **http://creativecommons.org/licenses/by-sa/2.5/**