CS601: Principles of Software Development

Javadoc.
JUnit Testing.
Immutability.

Olga A. Karpenko

Parts of this presentations are based on the materials of Prof. Engle.

Announcements

- Lab 2 is out, due on Friday
  - Look at documentation
  - Need to write tests + javadoc
  - >=5 commits
- Quiz on Wednesday

Javadoc

- Utility for writing/maintaining documentation
- Comments need to be written in special format
- HTML documentation is generated by Java

Javadoc Comments

- Provide for public classes or interfaces
- Provide for constructors and public methods

Javadoc tags

- `@author name`
- `@param paramName description`
- `@return description`
- `@throws exceptionName description`
- `@version release`
- `@deprecated text`
- `@see reference`

JUnit: Introduction

Levels of Testing

- Unit testing
  - Examines individual components
  - Create tests to try and break components
- Integration testing
  - Tests how modules integrate with each other
  - Tests parameter passing between modules

Levels of Testing

- System testing
  - Looks at the entire system
  - Captures errors resulting from unanticipated interactions between components

JUnit Testing

Unit Testing

- Tests individual methods
- Improves software quality
- Automatable
  - Code in test cases
  - Run tests every time you modify something
  - Helps catch if changes break something
- Performed by developers

JUnit

- Use JUnit 5.x or 4.x
  - Uses Java 5 annotations
  - Use @Before, @After, and @Test
- The provided jar file is for JUnit 4

JUnit

- Use @Before before a method to make it called before every test
  - Any setup that should happen before all tests
- Use @After before a method to make it called after every test
  - Any tear down that should happen after tests
- Use @Test to indicate a method is a unit test

JUnit

- Have one Assert method per test
  - assertTrue(), assertFalse()
  - assertNull(), assertNotNull()
  - assertSame(), assertNotSame()
  - assertEquals(), assertArrayEquals()
- Usually one assert method per unit test

JUnit Test Annotation

- Able to indicate an exception is expected in a unit test

  @Test(expected=IOException.class)

- Able to indicate a timeout if necessary

  @Test(timeout=1000)

- Using System.exit() will break unit tests

Best Practices

- One test class per class to be tested
- Tests for each method
- Make each test independent of others
- Name unit tests consistently
- Each unit test ~= one assertion
- Re-run tests whenever the code is changed

http://howtodoinjava.com/2012/11/05/unit-testing-best-practices-junit-reference-guide/

Naming Conventions

- http://osherove.com/blog/2005/4/3/naming-standards-for-unit-tests.html

JUnit tests

- Examples in IntelliJ*:
  - `StringUtil, StringUtilTest`
  - `UserRegistration, UserRegistrationTest`

* Note: The examples do not follow naming conventions

Testing Exceptions

- Does the method throw an appropriate exception when the input is invalid?

```
@Test(expected = NullPointerException.class)
public void testExceptions() {
  String s = StringUtil.concatenate(null, “ab”);
  fail(“NullPointerException isn’t thrown!");
}
```

http://tutorials.jenkov.com/java-unit-testing/exceptions.html

Testing Exceptions

- Does the method throw an appropriate exception when the input is invalid?

```
@Test
public void trimConcatenateNullException() {
 try{
  String s = StringUtil.trimConcatenate(null, "a");
  fail("NullPointerException isn't thrown!");
 }
 catch(NullPointerException e){
     // do nothing
 }
}
```

http://tutorials.jenkov.com/java-unit-testing/exceptions.html

Lab 2

- Write tests for:
  - shapes classes
  - comparator classes

References

- http://tutorials.jenkov.com/java-unit-testing/index.html

Immutability

The final Modifier

- Final Variables
- Final Methods
- Final classes

final Variables

- A final variable is not modifiable

```
final int num = 10;
```

- Must be initialized at the declaration OR at the constructors

final

- Declaration as final will prevent modification
  - Attempts to modify will be caught by compiler

final

- The meaning of “final” for Object Reference variables:

```
final List<Integer> arr = new ArrayList<>();
```

- Can we refer `arr` to another `ArrayList` object?
- Can we modify the object that `arr` is referencing?

final

- The meaning of "final" for Object Reference variables:

```
final List<Integer> arr = new ArrayList<>();
```

- Can we refer `arr` to another `ArrayList` object?
  - No
- Can we modify the object that `arr` is referencing?
  - Yes

Example

```
final List<Integer> arr = new ArrayList<>();
arr.add(42); // OK
arr = new ArrayList<Integer>(); // not OK
```

Constants

- Usually declared as a static final member
- Initialized outside of the constructor

```
public static final double PI = 3.1415;
```

final Methods

- Can not be overridden

final Class

- Can not be subclassed

Mutable and Immutable

- A mutable object:
  - can be modified after it's created
- An immutable object:
- Can not be modified after it's created
- Represents a single "value"

Immutable Objects

- Are Thread-safe
- Make good Map keys
- Easier to use

In-Built Immutable Classes

- String
- Integer
- Double
- Float
- Boolean

Strings

- Are immutable
- Can not be changed after creation
- “Modifying a string” = “creating a new string”

String Example

```
String s = ”HELLO";
s.toLowerCase();
System.out.println(s); // s did not change
```

String Example Modified

```
String s = ”HELLO";
s = s.toLowerCase();
System.out.println(s);  // s is changed
```

Example

```
String s1 =  “hi”;
String s2 =  “hi”;
```

- “Pool of strings”
- Both strings will point to the same object

StringBuffer

- Mutable
  - Contents can be changed!
- Is synchronized
- Has many utility methods
- There is an overhead

Example

```
public class StringBufferDemo{

  public static void main(String args[]){
      StringBuffer buffer =  new
  StringBuffer( “Java” );
      buffer.append( “Rocks” );
      System.out.println(buffer);
  }
}
```

StringBuilder

- Similar to the StringBuffer class
- Is not synchronized
- The overhead is less than using a StringBuffer
  - In a single threaded environment

Further Reading

- http://javarevisited.blogspot.com/2011/07/string-vs-stringbuffer-vs-stringbuilder.html

How to make a class immutable?

- Make all data fields private (and final)
- No mutator methods
- No accessor (get) methods can return a reference to a mutable data field
- Make the class final

Immutable Classes

- “Classes should be immutable unless there is a good reason to make them mutable”
  - Joshua Block (The author of “Effective Java”)