Leftovers

Last lecture and labs

Java code documentation

Java supports three types of comments. The first two are the // and the /* */.

The third type is called a documentation comment. It begins with the character sequence

/** and it ends with */.

Class description: add class description for each class

Note: before class definition and after all import statements

```
import java.io.*;
/** * This class demonstrates documentation comments.
* @author Tom Jones <address @ example.com>
* @version 1.2
* @since 2010-03-31 (the version of the package this class was
first added to)*/
public class SquareNum{ …
```

Method description: parameters and return type

```
/**
 * This method returns the square of num.
 *
 * This is a multiline description. You can use
 * as many lines as you like.
 *
 * @param num The value to be squared.
 * @return num squared.
 */
 public double square(double num) {
   return num * num;
 }
```

Testing your code for logical errors

- A method can fail for two reasons:
  - Logical error in its implementation.
  - Inability to obtain needed resource from environment. $\leftarrow$ Exception

Programming by contract

Programming style in which invocation of a method is viewed as a **contract** between client program and server classes, with each having explicitly stated responsibilities.

Documenting responsibilities

```java
/**
 * Create a new Explorer with the specified name,
 * initial location, strength, and tolerance.
 *
 * @require  strength >= 0
 *          tolerance >= 0
 *           name.length() > 0
 */
public Explorer (String name, int strength, int tolerance)
```

Programming by contract

- *Preconditions*: requirements on *client* of a method.
  - Labeled “require”

- *Postconditions:* requirements on *server* of a method.
  - labeled “ensure”

- Preconditions and postconditions are part of the contract.

Programming by contract

- For method invocation to be correct:
  - client must make sure that preconditions are satisfied at time of call.
  - If preconditions are satisfied, server guarantees that postconditions will be satisfied when method completes otherwise server *promises nothing at all*.

Programming by contract

- Consequence: test for every possible error condition only once.
  - Program efficiency.
  - Reduction of implementation complexity.

Programming by contract: example

- Complete specification of Explorer's constructor:

```
/**
 * Create a new Explorer with the specified name,
 * initial location, annoyability, and tolerance.
 *
 * @require  annoyability >= 0
 *           tolerance >= 0
 * @ensure  this.name().equals(name)
 *          this.location().equals(location)
 *          this.annoyability() == annoyability
 *          this.tolerance() == tolerance
 */
public Explorer (String name, Room location,
                    int annoyability, int tolerance)
```

Verifying preconditions

- Java's *assert* statement can be used in verifying preconditions.

```
assert booleanExpression ;
```

- The boolean expression is evaluated
  - if true, statement has no effect.
  - If false, statement raises an error condition stopping execution of program displaying cause of error.

Verifying preconditions

```
public Explorer (String name, Room location,
                    int annoyability, int tolerance) {
  assert annoyability >= 0;
  assert tolerance >= 0;

  this.name = name;
  this.location = location;
  this.annoyability = annoyability;
  this.tolerance = tolerance;
}
```

Pre-conditions summary

- Preconditions must be satisfied by client invoking method.
- Most often preconditions constrain values that client can provide as arguments when invoking method.
- Remember: if an argument is not constrained by a precondition, method must be prepared to accept *any value of the specified type*.

Example 1: constructor precondition java.util.Date class

```
/**
 * Create a new Date.
 * Arguments day, month, year must represent
 * a legal calendar date. Year must be > 1752.
 */
public Date (int day, int month, int year)
```

- Need to provide client a method to determine whether or not constructor arguments represent a legal date.
- Method is not a feature of a *Date* instance, but a utility of the *Date* class.

```
public static boolean isLegalDate (
      int day, int month, int year)
     Arguments day, month, year represent
     a legal calendar date.
```

Example 2: no preconditions

- Server promises to fulfill a contract only if client satisfies preconditions.

```
public int indexOf (Object item)
  The index of the first occurrence of the
  specified item on this List, or
  -1 if this List does not contain specified item.
```

- If we remove specification of returning a -1 if item not found in list, need to have precondition that item is in list.
- This puts an **unreasonable** burden on client.

Example 3: Valid indexes

- Assert statement used to verify preconditions.
- Two forms:

```
assert booleanExpression ;
assert booleanExpression : expression ;

//Interchange list.get(i) and list.get(j)
//    require  0 <= i, j < list.size()  …
private <Element> void interchange (
      List<Element> list, int i, int j) {
      assert 0 <= i && i < list.size():
       "precondition: illegal i";
      assert 0 <= j && j < list.size():
       "precondition: illegal j";
      …
```

Defensive programming

- Server validates all its arguments and notifies client with an exception if an argument is invalid.
- This programming style does not clearly delineate client and server responsibilities.
- Defensive programming results in
  - multiple checks of same conditions.
  - Code bloat
  - Misuse of exceptions to detect normal rather than exceptional conditions.

Postconditions

- Method `currentCount` in Counter is specified as

```
/**
 * The number of items counted.
 */
public int currentCount () { …
```

- Can be more precise : integer result is non-negative.

```
/**
 * The number of items counted.
 * @ensure this.currentCount() >= 0
 */
public int currentCount () { …
```

- *Postcondition:* condition that implementor promises will be satisfied when method completes execution.

Postconditions

- We can use **assert** to check post-conditions
- Postconditions often too complex to verify with simple conditions.
- Postconditions
  - can be tricky to handle;
  - often they involve comparing an object’s state *after* method execution to the object’s state *prior* to execution.
- Including such checks depends on where we are in the development process.

Running program with assertions

- Assertions are disabled by default
- To enable them use the following JVM argument:

```
java -ea mazeGame.game
```

$\uparrow$

Enable **A**ssertions

Distributing your application with Java WEB Start

- Allows to start your java application directly from the Internet using WEB browser
- Unlike Java applets, Web Start applications do not run inside the browser. However, by default they run in the sandbox. Only signed applications can be configured to have additional or even all permissions.
- Web Start has an advantage over applets in that it overcomes many compatibility problems with browsers' Java plugins and different JVM versions.
- On the other hand, Web Start programs are no longer part of the web page. They are independent applications that run in a separate thread

How to deliver your app through Java WEB start

- Go over your code and locate all places where app is using external resources, such as images, text files or sounds
- Add these external files directly to the package folder of the classes that use these resources (by dragging them into a package folder)
- Change how you access these external resources in your java classes

Step 1. Recode reading of all external resources

```
import java.net.URL;

//to load image from file:
try
{
                URL url = this.getClass().getResource("star.png");
                BufferedImage bi = ImageIO.read(url);
}
catch (IOException e) { e.printStackTrace(); }

//to load any input stream
try
{
                InputStream url = this.getClass().getResourceAsStream("surfingalien.mid");
                Sequence Seq = MidiSystem.getSequence(url);
}
catch (IOException e) { e.printStackTrace(); }
```

Step 2. Generate jar file for your project

- In Eclipse: Project -> export as jar file (non executable jar). Include all resources which are located in package directories
- Suppose we generated myproject.jar

Step 3. Generate *marina.jnlp* file

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp spec="1.0+" codebase="http://csci.viu.ca/~barskym/OOP2012/test1">
  <information>
    <title>My project</title>
    <vendor>VIU CSCI 331 team</vendor>
  </information>
  <resources>
    <!-- Application Resources -->
    <j2se version="1.6+" href="http://java.sun.com/products/autodl/j2se"/>
    <jar href="myproject.jar" main="true" />
  </resources>
  <application-desc
    name="My project"
    main-class="marinapackage.MainAppClass"
    width="800"
    height="600">
  </application-desc>
  <update check="background"/>
</jnlp>
```

Absolute path on server where your jar file is located

Name of the project jar file

Define main class which will start the application

Step 4. Create simple html file with Java WEB start link

Include deployJava script

```
<html>
 <head>
          <title> My project example</title>
          <meta charset="UTF-8" />
 </head>

 <body>
          <script src="http://www.java.com/js/deployJava.js"></script>
          <H1> Simple 2D animations </H1>

   <a href="javascript:deployJava.launchWebStartApplication('marina.jnlp','1.6.0');">
          My app</a>
 </body>
</html>
```