CSE 12
Abstraction, Classes, and Interfaces

- Software re-use
- The Inheritance and Composition Patterns
- Intro to UML
- Abstraction, continued
- ADT’s, Classes, and Interfaces
- Intro to the Java Collections Framework (JCF)
- Intro to Unit Testing with JUnit
- Review of exceptions
Software Reuse in Object-Oriented Programming

• You can write more software if you can make use of existing code

• This is the basic idea of software reuse

• Inheritance and Composition are two powerful design patterns for software reuse that are available in object-oriented programming languages

• Use them when appropriate!
Inheritance and Composition

• Inheritance defines an “is a” relationship between classes
  – A CheckingAccount “is a” BankAccount
  – A class *inherits* capabilities from its superclasses

• Composition defines a “has a” relationship
  – A BankAccount “has a” String to store the client’s name; a Date to store the date of the lastest transaction; etc.
  – Such a class is *composed* of one or more other classes, and can use their capabilities in its own implementation
Unified Modeling Language (UML)

• UML is a visual notation to describe the components of a software system and their relationships
• UML uses several kinds of diagrams to show these relationships
• In CSE 12 we will mainly use UML class diagrams: static pictures of some of the classes and interfaces in a system and their relationships to one another
• Some examples: UML class diagrams showing inheritance and composition relationships
UML Class Diagram: Inheritance

- SavingsAccount and CheckingAccount “is-a” BankAccount
- SavingsAccount and CheckingAccount inherit getBalance() and makeDeposit() methods from BankAccount. **Software reuse!**
A BankAccount “has-a” String, Date, and Address

BankAccount can use all the functionality of String, Date, and Address as needed to implement its own operations. Software reuse!
Collections

- A collection is an ADT that contains data elements, and provides operations on them
  - An instance of a collection is a data structure!
  - Since CSE 12 is a course in data structures, we will spend a lot of time on collections

- There are different kinds of collections, with different API’s and different performance tradeoffs

- The most common operations provided by a collection are adding, removing, and retrieving a data element from a collection… but there are others as well
Collection Operations

Most collections support these operations, but may give them different names

- Add an element
- Remove an element
- Determine if a collection contains an element
- Replace an element
- Determine how many elements a collection has
- Determine if a collection is empty
- Traverse a collection
- Determine if two collections are equal
- Clone (copy) a collection
- Serialize a collection
Abstraction (reviewed)

• A tool to manage complexity:
  • concentrate on the interface to a “thing” – *what the thing does*
  • while hiding its implementation details – *how it does it*

• Applying the concept of abstraction to data types leads to the idea of Abstract Data Types (ADT’s), which specify data values and operations while abstracting away from implementation details
ADT’s and Java interfaces

• Usually, instance variables should be *private* in a class
  • They are considered part of the implementation, not to be accessed directly from outside the class

• Then, we write *public* instance methods to manipulate the instance variables
  • *Mutator* methods change the values of instance variables; *Accessor* methods just ‘read out’ the values
  • This permits precise control over how the instance variables can be accessed and changed

• So, the principle of abstraction suggests we can concentrate on the public instance methods

• Java interfaces do that: they define a Java type, but specify only public instance method signatures (no method bodies, no instance variables)
Interfaces and Implementation

• A class implementing an interface defines an “is a” relationship between the class and the interface
  – A LinkedList “is a” List
  – A class inherits method signatures from interfaces it implements

• An interface can extend one or more other interfaces
  – A Set “is-a” Collection
  – An interface inherits method signatures from interfaces it extends
• LinkedList and ArrayList “is-a” List

• List is a datatype, but you cannot create a List by new List()

• To create a List, you must create an instance of a class that implements the List interface
The Java Collections Framework

• Different programming language libraries provide various data structures that can be used by application programmers

• Java’s data structure library is referred to as the Java Collections Framework (JCF)

• The JCF is a collection of interfaces, abstract and concrete classes providing a standard set of collection (data structure) types, in the java.util package

• The relationship among interfaces and classes in the JCF can be shown with UML diagrams…
Some Interfaces in JCF
Some Classes Implementing Interfaces in JCF
Properties of good software

• These are properties of good software:
  
  – **robustness**: can the program detect exceptional conditions and deal with them or shutdown gracefully?
  
  – **correctness**: does the program do what it is supposed to do, according to its specification?
  
  – **efficiency**: all programs use resources (time and space and energy, i.e. CPU cycles and memory and battery life); does the program use no more resources than necessary?
Software Testing

- Software testing is important, to check correctness (is the software doing what it is supposed to do, according to its specification?)

- In CSE 12 we will take a test-driven development approach, with unit testing

- **Unit testing** means: testing every individual ‘unit’ of the software system
  - In object-oriented programming, a unit is a *method*

- **Test-driven development** means: first writing tests, and then writing the software that will be tested
  - This inspires you to understand the requirements of the software before writing it, and leads to better software
A Flowchart for Test-driven Development
Black-box Testing

- Treat a class and its methods as "black boxes"
  - From the specification, you know for a given input what should be output by the “box”…
  - … but you don’t how it was generated because you can’t see the implementation details inside the “box”!
- A test case says what should happen. You can think of it in 4 parts:

  Format for a test case

<table>
<thead>
<tr>
<th>Operation</th>
<th>Purpose</th>
<th>Object State</th>
<th>Expected Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>A call to an operation of the ADT to be tested.</td>
<td>The role the operation plays in the test case.</td>
<td>The expected state of the test object once the operation is complete.</td>
<td>The expected output (if any).</td>
</tr>
</tbody>
</table>
Clear-box Testing

• (Also known as “white-box testing”)
• If you can look inside the black box and see how a method is implemented, you can do more detailed testing.
• For example, you can do path coverage testing: test all possible paths through the code

```java
stmt0;
if ( a < 0 ) {
  stmt1;
  stmt2;
} else {
  stmt3;
  stmt4;
}
stmt5;
```
Unit testing

- Whether you are doing black-box or clear-box testing, you should test every important unit of a software system.

- What is a unit? In object-oriented programming, usually a software unit is taken to be a single method.

- So: we should test every method of every class in the software.

- JUnit is a widely used framework for unit testing of Java software...
Unit testing with the swingui of Junit

3.8.1: all tests passed

The green bar of happiness; all tests passed!

Names of the testing methods corresponding to the test cases you prepared. A green check mark means the test passed.
When one or more tests fail

The red bar of sadness; some tests failed

Names of the testing methods in your test suite. A red X means the test failed.

Stack trace telling what was expected, what was generated and where the test failed; very handy!
JUnit basics

• To do unit testing in the Junit framework:
• Define a subclass of `junit.framework.TestCase`
• Optional:
  – Define instance variables that store the state of the “test fixture”, i.e. the objects that will be tested
  – Initialize the fixture state by overriding the `setUp()` instance method
  – Clean-up the fixture state after a test by overriding the `tearDown()` instance method
• Define public void no-argument methods with names that start with `test`. Each “testXXX” method should be written to test a particular aspect of the test fixture
• Define a `main()` method to run your TestCase class as a program
TestCase and test fixtures

• Define a subclass of `junit.framework.TestCase`
• Define instance variables that store the state of the “test fixture”, i.e. the objects that will be tested

```java
import junit.framework.*;

public class RectangleTester extends TestCase {
    private Rectangle r1, r2; // test fixtures
```
setUp() and tearDown()

/* Called AUTOMATICALLY before each testXXX() method is run */
protected void setUp() {
    r1 = new Rectangle();
    r2 = new Rectangle(2.0,3.0);
}

/* Called AUTOMATICALLY after each testXXX() method is run */
protected void tearDown() {
    r1 = null;
    r2 = null;
}

setup();
testXXX();
tearDown();

This is the sequence of calls the JUnit framework does for you automatically for each test method testXXX() that is invoked.

Make sure each test method starts with a clean copy of the test fixture.
/** Test case 2.1: verify that default constructor sets default instance variable values correctly */
public void testDefaultInstance() {
    assertEquals(1.0, r1.getLength());
    assertEquals(1.0, r1.getHeight());
}

/** Test case 2.6: verify that mutator for length throws exception on illegal input. */
public void testMutatorIllegalInput() {
    try {
        r1.setLength(0); // 0 is illegal, should throw fail();
    } catch (IllegalArgumentException e) {
        // test passes
    }
}
Running your TestCase class

To run a class as a program, the class must have a `public static void main()` method. Here are two ways to define it in the Junit 3.8.1 framework, depending on whether you want a GUI or text version:

```java
/** Run RectangleTester as a gui application */
public static void main(String args[]) {
    junit.swingui.TestRunner.main(new String[] {"RectangleTester"});
}

/** Run RectangleTester as a text console application */
public static void main(String args[]) {
    junit.textui.TestRunner.main(new String[] {"RectangleTester"});
}
```
More JUnit basics

- Test fixture instance variables are optional: you can do the same thing with local variables inside the test methods
- If you are not using test fixture instance variables, you do not need to define setUp() and tearDown() either
- When you run your TestCase class as a program, each “testXXX” method is called automatically…
- If a Junit assertion fails when a testXXX method runs, that test fails
- If testXXX method throws an exception, that is considered an error, not a test failure!
- If a testXXX method returns normally, that test passes
JUnit 3.8.1 assertion methods

• The JUnit framework provides many useful assertion methods to use in your testXXX() methods

assertEquals(x, y)    // fail if x is not equal to y
assertTrue(b)         // fail if b has boolean value false
assertFalse(b)        // fail if b has boolean value true
assertSame(x, y)       // fail if x and y point to different objects
assertNotSame(x, y)    // fail if x and y point to the same object
assertNull(x)          // fail if x is not null
assertNotNull(x)       // fail if x is null
fail()                 // always fails

• All these assertion methods are overloaded with a version that takes an additional first argument of type String, a message which will be printed out in some contexts
Specifying an ADT

- An ADT specifies the possible values instances of the ADT can have, and specifies the operations the ADT provides on those values.
- In more detail, an ADT specification should generally include these:

  - **ADT name:** The name of the data type.
  - **ADT description:** A brief summary description of the type.
  - **ADT invariants:** Assertions that must always be true of any instance of this type.
  - **ADT attributes:** Aspects of the state of an instance of the type, as observable by a client.
  - **ADT operations:** The behavior of an instance of the type, as observable by a client.
Specifying ADT operations is particularly important: these operations constitute the API, and are the primary way that users of the ADT interact with its instances.

For each ADT operation, you should specify:

- **responsibilities**: A brief summary of what the operation does.

- **pre-conditions**: What must be true on entry to the operation if the operation is to execute successfully. This may include assumptions about the state of the object, assumptions about the parameters passed in, etc. An operation's pre-conditions must be consistent with all of the ADT's invariants.

- **post-conditions**: What the operation guarantees to be true when it returns, if the pre-conditions were true when the operation was called. An operation's post-conditions must be consistent with all of the ADT's invariants.

- **returns**: The value, if any, returned by the operation.

- **exceptions**: A description of the exceptions the operation may throw.
Programming by contract

• A precise and complete specification of an ADT operation is a *contract* for the operation
  – It lays out the responsibilities of the *provider* (the method) and the *client* (any caller of the method)
  – If both sides meets their responsibilities, the operation will work
  – The contract is a guide both for the implementer and the users of the method

• Writing these specifications first, then implementing them, is sometimes called *programming by contract*
Pre- and Post-conditions

- Think of method preconditions as the method saying: “I require this.”
- Think of method postconditions as the method saying: “I ensure this.”

- If the preconditions are satisfied when the method is called, then the method guarantees that the postconditions will be satisfied when the method returns.
- If the preconditions are not satisfied -- that is, if the client does not meet its end of the contract -- then the method does not guarantee anything!
Checkpoint

• What is the relationship between ADT or class invariants and an operation’s pre- and post-conditions?

• How do pre- and post-conditions relate to writing a test for the operation?

• Can you write a test for what the operation does when its pre-conditions are not met?
Example ADT specification: Rectangle

• Description
  – A rectangle is a four-sided shape in which opposite sides are parallel and equal in size. The size of a side must be greater than 0. A rectangle has four right angles.

• Invariants
  – Opposite sides are of equal size.
  – Length and height must be greater than 0.

• Attributes
  – DEFAULT_SIZE a constant, the default size for the dimensions
  – length size of the “top” and “bottom” sides
  – height size of the “left” and “right” sides
  – surface area the surface area of the rectangle
  – perimeter the perimeter of this rectangle
Review of Java Exceptions

• Exceptions are a way of signaling that something exceptional has happened at runtime

• When an “exceptional condition” occurs, an exception object is created storing information about the nature of the exception (kind, where it occurred, etc.), and then this exception object is *thrown*

• The JVM looks back through the chain of method calls for a block of code to *catch* and handle the exception

• If there isn't a catch block for the exception, your program will be killed, and some diagnostic info printed to the terminal
Java’s Exception Hierarchy

remember, this means extends

These are some of the more useful methods defined in Throwable that we will be using and which are inherited by all its descendants.
Kinds of exceptions

- **Checked exceptions** –
  - Descended from class Throwable, but not descended from RuntimeException or Error.
  - The compiler will check that you either catch or rethrow checked exceptions.

- **Unchecked exceptions** –
  - Descended from RuntimeException or from Error.
  - The compiler *does not* check to see that you catch or explicitly rethrow these exceptions (though of course you can catch or declare them if you want).
Creating and Throwing an Exception

• Though it is not strictly required by the Programming-by-Contract paradigm, a method can check to see if its own preconditions are met
• If it finds that its preconditions are not met, what should it do?
• Creating and throwing an exception, instead of attempting to complete the operation or returning a value, is often a good design decision
• If so, this should be documented as part of the specification of the method
• Example: setLength() method in a Rectangle class…
Creating and Throwing an Exception

• Two ways to specify the setLength() operation…

setLength( newLength )
pre-condition: newLength > 0
responsibilities: sets length to newLength
post-condition: rectangle’s length field is set to newLength
returns: nothing

• Becomes:

setLength( newLength )
pre-condition: newLength > 0
responsibilities: sets length to newLength
post-condition: rectangle’s length field is set to newLength
returns: nothing
throws: IllegalArgumentException if precondition is not met
Implementing setLength

/**
 * Set the length dimension of this <tt>Rectangle</tt>.
 * @param theLength the new length of this <tt>Rectangle</tt>;
 * @throws IllegalArgumentException if <tt>theLength</tt> is <= 0.
 */
public void setLength( double theLength ) {
    if ( theLength <= 0 )
        throw new IllegalArgumentException("Illegal Rectangle length (" +
            theLength + "): must be > 0 ");
    this.length = theLength;
}
Next time

- Test plans, test cases, and more about JUnit
- Generic programming and Java generic types
- The Collection and List Interfaces in the JCF
- Iterators and the Iterator Pattern
- Developing a test plan for a Collection or List class

Reading: Gray, Ch 4 and Ch 5