CSE 12 – Basic Data Structures
Prof. Christine Alvarado
[Slides borrowed/adapted from slides by Cynthia Lee]
Announcements

1. Account problems? Contact ACMS
2. Gradesource report linked. Check your participation score! (Quiz scores coming soon)
Today’s Topics

1. Intro to UML
   - Inheritance vs Composition
2. ADT’s, Classes, and Interfaces
3. Junit and testing
ADTs vs specific implementations

- Sometimes the line between the abstract data type and the implementation is confusing. E.g. Car brakes

To drive my car, I rely on the abstract idea of brakes.

But for that idea to work, my car needs to have a specific implementation of brakes.

Someone needs to care about both sides!
ADTs vs specific implementations

- It usually depends on which side you’re looking from. E.g. Car brakes

In this class (and as a computer scientist) you will be the driver AND the mechanic!
Review: Collections and Presents (in Java)

This Bag is a Collection. A Collection is an ADT that specifies how you can interact with a group of objects (e.g., count them, put one in, etc).

The Bag itself provide the specific implementation of the Collection interface. It contains the code that actually does the counting, adding etc.
Review: Collections and Presents (in Java)

<<interface>>

Collection

add(Object)

size()

Abstraction

Implementation Details (many options!)
Expressing Implementation in UML

A Collection is an abstract data type (ADT) provided by Java in the form of an interface. Generally in Java interfaces are used to express ADTs.

A Bag is a class that we write that implements the Collection interface.* A Bag “is-a” Collection.

*In general, you would never directly implement the Collection interface in Java. More on this later.
A List is another ADT expressed as an interface in Java.

- 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.
But what’s in the Bag/List? Java Generics

- Key to Java’s Collections are **generics**
  - Generics answer the question: “what is the collection a collection of?”
  - If this question isn’t enforced with generics, there can be runtime errors when the what comes out of the collection is not what you expected
- `List<E>` means a list of things that are of type E.
Which choice is the most reasonable model of our bag of presents?

```java
public class Bag<Present>
    implements Collection<Present>
```

Java's Collection interface uses generic types:
```java
Collection<E> Where E can be any type when we create a collection.
```

We are “instantiating” that type as “Present” for our class Bag so that the bag can only hold Presents.

(We could also create a “generic” Bag that could hold any kind of object)

**IMPORTANT:** The Bag is still a concrete instantiation, not an ADT, even if it’s allowed to hold different types
Placing limits on what kind of thing our collection can store

- This is a kid, so we ONLY want to ensure the Presents are:
  - **Wrapable** (can be wrapped in paper)
    - Implements a “wrap()” method
  - **Playable** (can be played with)
    - Implements a “play()” method

- How can we represent these requirements in Java?
How can we represent the Playable and Wrapable requirements in Java?

1. Create interfaces Playable (has a method `void play()`) and an interface Wrapable (has a method `void wrap()`)
2. Now how do we use those interfaces to impose the requirements on Present?

A. Create interfaces Playable and Wrapable

B. Present

C. Other/ none/ more than one

D. I don’t understand this!!
Back to the presents example: How can we represent the Playable and Wrapable requirements in Java?

1. Create interfaces Playable and Wrapable

2. Create a class* Present that implements Wrapable and Playable.

3. Create classes for different kinds of presents, all of which extend* Present

4. Now objects of those classes can be stored in the Bag<Present> collection!
Some Interfaces in JCF
Some Classes Implementing Interfaces in JCF
Implementations Exist

- For many/all of the data types we’ll study
- Sometimes we’ll have you use pre-built classes from the Java Collections Framework
- Sometimes we’ll have you build (or extend) these classes yourself
- The BEST way to really understand these data types is to program them at least once yourself.
Software Testing
Test-driven Development

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

- Test-driven development means: first writing tests, and then writing the software that will be tested.

WARNING

“RELIGIOUS” WARS BREAK OUT OVER WHEN IN THE DEVELOPMENT CYCLE TO DEFINE TESTS
(MOST AGREE ON THIS: “DO NOT LEAVE IT AS THE LAST THING TO DO”)
Unit testing

(Refer to the reading)

• 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!
public class Counter {

    private int count = 0;
    private int step = 1;

    public Counter() {
        // Nothing to do here
    }

    public Counter(int theStep) {
        int s;
        if ( (s = Math.abs(theStep)) > 0) {
            step = s;
        }
    }

    public int getCount() {
        return count;
    }

    public void increment() {
        count += step;
    }

    public void decrement() {
        if (count > step) {
            count -= step;
        } else {
            count = 0;
        }
    }

    public void reset() {
        count = 0;
    }
}

How many unit tests would you write to test this class?
A. 1  B. 2  C. 4  D. 5  E. More than 5
import junit.framework.TestCase;

public class CounterTest extends TestCase {

    private Counter counter, counter2;

    @Override
    /* this sets up the test fixture. JUnit invokes this method 
     * before every testXXX method */
    protected void setUp() throws Exception {
        super.setUp();
        counter = new Counter();
        counter2 = new Counter(2);
    }

    public void testDefaultValueOfCounterIsZero() {
        System.out.println("Checking Default Counter Value is Zero");
        assertEquals(0, counter.getCount());
        assertEquals(0, counter2.getCount());
    }

    Discussion: Why do we have 2 counters and two assert statements in testDefaultValueOfCounterIsZero()?
In HW1 you are given the class Counter and you will write a Unit tester for it. Is this an example of Black Box or Clear Box testing?
A. Black Box
B. Clear Box
C. It could be either
D. Neither
**setUp() and tearDown()**

/* Called AUTOMATICALLY before each testXXX() method is run */
protected void setUp() {
    super.setUp();
    counter = new Counter();
    counter2 = new Counter(2);
}

/* Called AUTOMATICALLY after each testXXX() method is run */
protected void tearDown() {
    counter = null;
    counter2 = null;
}

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

Make sure each test method starts with a clean copy of the test fixture.

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

See HW1, problem 1 to explore how to run tests without a main method! (Command line, Eclipse... and as a bonus try Dr. Java!)
More details on JUnit: See the reading!

Also, you’ll need to rely on Java’s documentation for its LinkedList class. We’ll talk a lot about LinkedLists, but we want you to try using them first.
Next time

- Developing an ADT
- The Collection and List Interfaces in the JCF
- Iterators and the Iterator Pattern