CSE 143
Lecture 6

interfaces; eclipse; testing

slides adapted from Marty Stepp
http://www.cs.washington.edu/143/

Interfaces (9.5)

- **interface**: A list of methods that a class can promise to implement.
  - Inheritance gives you an is-a relationship *and* code sharing.
    - A `Lawyer` can be treated as an `Employee` and inherits its code.
  - Interfaces give you an is-a relationship *without* code sharing.
    - A `Rectangle` object can be treated as a `Shape` but inherits no code.
  - Analogous to non-programming idea of roles or certifications:
    - "I'm certified as a CPA accountant.
      This assures you I know how to do taxes, audits, and consulting."
    - "I'm 'certified' as a Shape, because I implement the Shape interface.
      This assures you I know how to compute my area and perimeter."

Interface syntax

```
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
    ...
    public type name(type name, ..., type name);
}
```

Example:

```
public interface Vehicle {
    public int getSpeed();
    public void setDirection(int direction);
}
```

Interface syntax

```
public class name implements interface name{
    …
}
```

Example:

```
public class Car implements Vehicle {
    …
    public int getSpeed() {
        return speed;
    }
    public void setDirection(int direction) {
        this.direction = direction;
    }
}
```

Writing testing programs

- Some programs are written specifically to test other programs.
- If we wrote `ArrayIntList` and want to give it to others, we must make sure it works adequately well first.
- Write a client program with a `main` method that constructs several lists, adds elements to them, and calls the various other methods.

Tips for testing

- You cannot test every possible input, parameter value, etc.
  - Even a single `(int)` method has 2^32 different possible values!
  - So you must think of a limited set of tests likely to expose bugs.

- Think about boundary cases
  - positive, zero, negative numbers
  - right at the edge of an array or collection's size

- Think about empty cases and error cases
  - 0, -1, null; an empty list or array
  - an array or collection that contains null elements

- Write helping methods in your test program to shorten it.

More testing tips

- Focus on **expected** vs. **actual** behavior

- the test shouldn't just call methods and print results; it should:
  - call the method(s)
  - compare their results to a known correct expected value
  - if they are the same, report that the test "passed"
  - if they differ, report that the test "failed" along with the values

- test behavior in combination
  - maybe `add` usually works, but fails after you call `remove`
  - what happens if I call `add` then `size`? `remove` then `toString`?
  - make multiple calls; maybe `size` fails the second time only

Example `ArrayIntList` test

```
public static void main(String[] args) {
    int[] a1 = {5, 2, 7, 8, 4};
    int[] a2 = {2, 7, 42, 8};
    int[] a3 = {7, 42, 42};
    helper(a1, a2);
    helper(a2, a3);
    helper(new int[] {1, 2, 3, 4, 5}, new int[] {2, 3, 42, 4});
}

public static void helper(int[] elements, int[] expected) {
    ArrayIntList list = new ArrayIntList(elements);
    for (int i = 0; i < elements.length; i++) {
        list.add(elements[i];
    }
    list.remove(0);
    list.remove(list.size() - 1);
    list.add(2, 42);
    for (int i = 0; i < expected.length; i++) {
        if (list.get(i) != expected[i]) {
            System.out.println("fail; expect " + Arrays.toString(expected)
                               + ", actual " + list);
        }
    }
}
```