CSE 331
Software Design & Implementation
Section: Sets; JUnit; AFs
Reminders

• HW4 is challenging for many students. Start early!

Upcoming Deadlines

• HW3 due 11pm tonight (7/07)
• Prep. Quiz: HW4 due 11pm Tuesday (7/11)
Last Time...

- Abstract Data Types (ADTs)
- Representation Invariants
- Abstraction Functions

Today’s Agenda

- Review: AFs
- JUnit
- Sets and HW4
Abstract Data Types (ADTs)

• Abstraction representing some set of data
  – Meant to express the meaning/concept behind some Java class

• Different from implementation/Java fields!
  – Same ADT can have many different implementations
Abstract data types by example

Review ADT concepts through two examples:

• A Line ADT
• A Rectangle ADT

On the course website, see “Resources” → “Class and Method Specifications” for a handy guide with full details.
Abstraction Functions (AFs)

- Let’s say we have an ADT
  - And we choose some way to implement it

- How does the concrete implementation relate to our ADT?

- This is an **abstraction function**
  - Maps object implementation (our Java fields) to the abstract state
  - Ex: “How does a Triangle object from Triangle.java represent a Triangle ADT?”
  - Note: specific to implementation
Diagram

ADT specification

Abstract States

Abstraction Barrier

Fields in our Java class

Abstraction function (AF): Relationship between ADT specification and implementation
Line ADT

Concept: A line segment in the Cartesian co-ordinate plane
/**
 * A Line is a mutable 2D line segment with endpoints p1 and p2.
 */

public class Line {
    ... // rep invariant, fields, methods, etc.
}
Line ADT: Representation #1

/**
 * A Line is a mutable 2D line segment with endpoints
 * p1 and p2.
 */
public class Line {
    // Abstract state is ___
    private Point p1, p2;
}

What is our abstraction function?
Line ADT: Representation #1

/**
 * A Line is a mutable 2D line segment with endpoints
 * p1 and p2.
 */
public class Line {
    // Abstract state is line with endpoints p1 and p2
    private Point p1, p2;
}
Line ADT: Representation #2

/**
 * A Line is a mutable 2D line segment with endpoints
 * p1 and p2.
 */
public class Line {
    // Abstract state is _____
    private int x1, x2;
    private int y1, y2;
}

What is our abstraction function?
Line ADT: Representation #2

/**
 * A Line is a mutable 2D line segment with endpoints
 * p1 and p2.
 */

public class Line {
    // Abstract state is line with endpoints (x1, y1) and
    // (x2, y2)
    private int x1, x2;
    private int y1, y2;
}

Does this representation have any advantages over #1?
/**
 * A Line is a mutable 2D line segment with endpoints
 * p1 and p2.
 */

class Line {
    // Abstract state is ____
    private int x1, y1;
    private double angle;
    private double len;
}
Line ADT: Representation #3

/**
 * A Line is a mutable 2D line segment with endpoints
 * p1 and p2.
 */
public class Line {
    // Abstract state is line with endpoints (x1, y1) and
    // (x1 + len * cos(angle), y1 + len * sin(angle))
    private int x1, y1;
    private double angle;
    private double len;
}

Does this representation have any advantages over #1?
Try it yourself!

Write your own specification of a Rectangle ADT on the handout.

Then give two different possible representations for your Rectangle ADT and write abstraction functions for them.
Testing: A quick introduction

• In past assignments, you have run the test suite.

• But now you must start writing your own tests!
JUnit

- Industry-standard Java toolkit for unit testing
  - We're using JUnit 4.12
  - Check out the javadocs

- A unit test is a test for one “component” by itself
  - “Component” typically a class or a method

- Each unit test written as a method
  - We'll see the particulars in a moment...

- Closely related unit tests should be grouped into a class
  - For example, all unit tests for the same ADT implementation
Writing tests with JUnit

A method annotated with `@Test` is flagged as a JUnit test

```java
import org.junit.*;
import static org.junit.Assert.*;

/** Unit tests for my Foo ADT implementation */
public class FooTests {
    @Test
    public void testBar() {
        ... /* use JUnit assertions in here */
    }
}
```
Using JUnit assertions

- JUnit assertions establish success or failure of the test method
  - Note: JUnit assertions are different from Java’s assert statement

- Use to check that an actual result matches the expected value
  - Example: `assertEquals(42, meaningOfLife());`
  - Example: `assertTrue(list.isEmpty());`

- A test method stops immediately after the first assertion failure
  - If no assertion fails, then the test method passes
  - Other test methods still run either way

- JUnit results show details of any test failures
Common JUnit assertions

JUnit’s documentation has a full list, but these are the most common assertions.

<table>
<thead>
<tr>
<th>Assertion</th>
<th>Failure condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>assertTrue(test)</td>
<td>test == false</td>
</tr>
<tr>
<td>assertFalse(test)</td>
<td>test == true</td>
</tr>
<tr>
<td>assertEquals(expected, actual)</td>
<td>expected and actual are not equal</td>
</tr>
<tr>
<td>assertSame(expected, actual)</td>
<td>expected != actual</td>
</tr>
<tr>
<td>assertNotSame(expected, actual)</td>
<td>expected == actual</td>
</tr>
<tr>
<td>assertNotNull(value)</td>
<td>value != null</td>
</tr>
<tr>
<td>assertNotNull(value)</td>
<td>value == null</td>
</tr>
</tbody>
</table>

Any JUnit assertion can also take a string to show in case of failure, e.g., assertEquals(“helpful message”, expected, actual).
Always* use >= 1 JUnit Assertion

- If you don’t use any JUnit assertions, you are only checking that no exception/error occurs

- That’s a pretty weak notion of passing a test; rarely the best test you could write

- Having more than one JUnit assertion in a test may make sense, but one is the most common scenario

* Special case coming in a couple slides
JUnit assertions vs Java’s assert

• Use JUnit assertions only in JUnit test code
  – JUnit assertions have names like `assertEquals`, `assertNotNull`, `assertTrue`
  – Part of JUnit framework used to report test results
    • Accessed via `import org.junit`....
  – Don’t use in ordinary Java code (never `import org.junit`.... in non-JUnit code)

• Use Java’s `assert` statement in ordinary Java code
  – Use liberally to annotate/check “must be true” / “must not happen” / etc. conditions
  – Use in `checkRep()` to detect failure if problem(s) found
  – Do not use in JUnit tests to check test result – does not interact properly with JUnit framework to report results
Checking for a thrown exception

- Should test that your code throws exceptions as specified

- This kind of test method fails if its body does not throw an exception of the named class
  - May not need any JUnit assertions inside the test method unlike our previous guideline

```java
@Test(expected=IndexOutOfBoundsException.class)
public void testGetEmptyList() {
    List<String> list = new ArrayList<String>();
    list.get(0);
}
```

- **Do not** use `assertThrows()` (that comes in JUnit 4.13, and we are using JUnit 4.12)
JUnit does not promise to run tests in any particular order.

However, JUnit can run helper methods for common setup/cleanup
• Run before/after *each* test method in the class:

```java
@Before
public void m() { ... }

@After
public void m() { ... }
```

• Run once before/after running *all* test methods in the class:

```java
@BeforeClass
public static void m() { ... }

@AfterClass
public static void m() { ... }
```
Demo: JUnit Tests Example

Now let’s look at some example JUnit tests...
Tips for effective testing

• Use constants instead of hard-coded values
  – Makes easier to change later on

• Take advantage of assertion messages

• Give a descriptive name to each unit test (method)
  – Verbose but clear is better than short and inscrutable
  – Don’t go overboard, though :-)

• Write tests with a simple structure
  – Isolate bugs one at a time with successive assertions
  – Helps avoid bugs in your tests too!

• Aim for thorough test coverage
  – Big/small inputs, common/edge cases, exceptions, ...
Worksheet: Test Design

- Work in small groups
- Give logic of the tests, not actual code
- Only test the operations provided on the worksheet
- More details in lecture if additional information/review needed
Before next lecture...

1. Do HW3 tonight! (reminder: deadline is 11pm)
   - Written portion (submit PDF on Gradescope)
   - Coding portion (push and tag on GitLab)

2. Start HW4 Written early – many find it challenging!

3. Review JUnit testing slides discussed in this section.

HW4 Background: Floats

- Floats vs. Doubles
  - Both represent floating point numbers, but doubles are twice the size (think \texttt{int} vs \texttt{long})
  - But we will be using \texttt{floats}

- Special cases:
  - \texttt{Float.POSITIVE_INFINITY} and \texttt{Float.NEGATIVE_INFINITY}
  - \texttt{Float.NaN} – means not a number

- Operations where either one of the operands is \texttt{NaN}
  - \textbf{All operations} will return \texttt{NaN}
  - e.g. \texttt{NaN * 1.23456f = NaN}

- Including \texttt{==}
  - \texttt{Float.NaN == Float.NaN} \texttt{-> false}
  - Use \texttt{Float.isNaN()} or \texttt{Float.isFinite()} instead
Finite Sets

• In HW4, we will be working in the `FiniteSet` class, which represents a set of points along a number line, where each point is a `float`.

• Let’s say we choose to represent this as an array of floats, i.e. `float[]`
• We need to make some choices:
  – Should we allow duplicates? Why or why not?
  – Should we sort our array? Why or why not?

• We will not allow duplicates and keep the array sorted.
• We will also store a `Float.NEGATIVE_INFINITY` as the first element in the array and a `Float.POSTIVE_INFINITY` as the last element...
  – This will make reasoning about it easier. For instance, we can guarantee that there is an index \( i \) such that \( D[i] < x < D[i+1] \)
FiniteSet Field

private final float[] vals;

The set \{ -5.3, 1.48, 7.1234, 463.8 \} will be represented as:
[Float.NEGATIVE_INFINITY, -5.3, 1.48, 7.1234, 463.8, Float.POSITIVE_INFINITY]

What is our representation invariant and abstraction function?

// Points are stored in an array, in sorted order, with an
// extra -infinity at the front and +infinity at the end
// to simplify union etc.
//
// RI: -infinity = vals[0] < vals[1] < ... <
// vals[vals.length-1] = +infinity
// AF(this) = { vals[1], vals[2], ..., vals[vals.length-2] }
Some common set operations:

- Finding the **union** (\(\cup\)) of set A and set B. This is a **new** set of points that are **either** in A, B, or **both** A and B:
  
  ```
  \text{union}([-\infty, 1, 4, 5, 7, \infty], [-\infty, 1, 6, 7, 11, \infty])
  = [-\infty, 1, 4, 5, 6, 7, 11, \infty] \Rightarrow \{ 1, 4, 5, 6, 7, 11 \}
  ```

- Finding the **intersection** (\(\cap\)) of set A and set B. This is a **new** set of points that are in **both** A and B:
  
  ```
  \text{intersection}([-\infty, 1, 4, 5, 7, \infty], [-\infty, 1, 6, 7, 11, \infty])
  = [-\infty, 1, 7, \infty] \Rightarrow \{ 1, 7 \}
  ```

- Finding the **difference** (- or \(\setminus\)) of set A and set B. This is a **new** set of points that are **in** A but **not** B:
  
  ```
  \text{difference}([-\infty, 1, 4, 5, 7, \infty], [-\infty, 1, 6, 7, 11, \infty])
  = [-\infty, 4, 5, \infty] \Rightarrow \{ 4, 5 \}
  ```
For much of the assignment, you will be working in SimpleSet.java

- A SimpleSet is defined as either a finite set of points or the complement of a finite set of points (meaning everything but).
  - e.g. given the set of points \( \{1, 7, 9\} \):
    - we can have a simple set that contains 1, 7, and 9 or
    - one that contains all real numbers except 1, 7, and 9

    /**
    * Represents an immutable set of points on the real line that is easy to
    * describe, either because it is a finite set, e.g., \([p_1, p_2, \ldots, p_N]\),
    * or because it excludes only a finite set, e.g., \(\mathbb{R} \setminus \{p_1, p_2, \ldots, p_N\}\).
    * As with FiniteSet, each point is represented by a Java float with a
    * non-infinite, non-\(\text{NaN}\) value.
    */

    public class SimpleSet {
Let’s now skim the starter code...