



# CSE332: Data Abstractions

## Section 4

Hyeln Kim

CSE 331 Slides

Spring 2013

# Section Agenda

---

- **Project 2: Shake-n-Bacon**
  - More Generics
  - Comparator
  - Inheritance review
  - Iterator / Anonymous class
  - JUnit Testing & Other Debugging tools
- Project1 Feedback
- B-Tree, HW2, HW3 question

---

# Project 2

Shake-n-Bacon

# Word Frequency Analysis

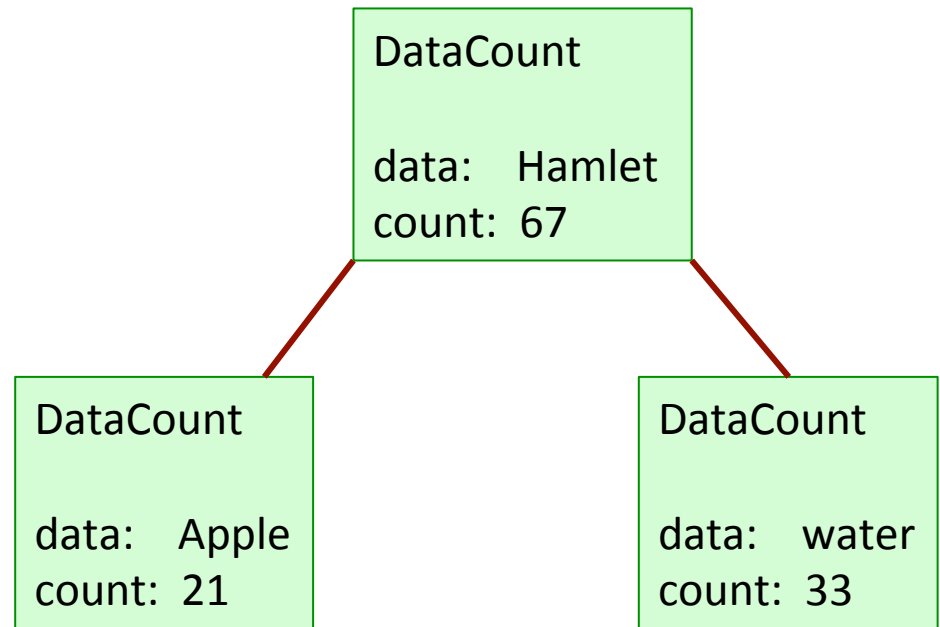
---

- **Phase A:** Implement 3 ADTs
  - Due next Wednesday
  - Word frequency analysis using different DataCounters

- **AVLTree**

- **MoveToFrontList**

- **FourHeap**



---

# Generics

Generic Arrays & Wildcard

# Generic Arrays

---

- **Field & variable can have generic array type**

```
E[] elemArray;
```

- **Cannot create new generic array**

```
E[] elemArray = new E[INITIAL_CAPACITY]; // Error
```

- Arrays need to “know their element type”
- Type “E” is unknown type

- **Workaround with Object[] - Unavoidable warning**

```
E[] elemArray = (E[]) new Object[INITIAL_CAPACITY];  
// Generates warning, but ok
```

# Array of Parameterized type

- **Cannot create array of parameterized type**

```
DataCount<E>[] dCount = new DataCount<E>[SIZE]; //Error
```

- **Object[] does not work - ClassCastException**

```
DataCount<E>[] dCount = (DataCount<E>)  
                        new Object[SIZE]; //Exception
```

- Arrays need to “know their element type”
- Object not guaranteed to be DataCount

- **Specify it will always hold “DataCount”**

```
DataCount<E>[] dCount = (DataCount<E>)  
                        new DataCount[SIZE]; // ok
```

# Generics & Inner Class

- **Do not re-define type parameter**

```
class OuterClass<E> {  
    class InnerClass<E> {}  
} // No ☹️
```

```
class OuterClass<E> {  
    class InnerClass {}  
} // Yes 😊
```

- Works, but not what you want!!

- Analogous of local variable shading field

```
class SomeClass {  
    int myInt;  
    void someMethod() {  
        int myInt = 3;  
        myInt ++;  
    } // Not the field  
}
```

```
class OuterClass<E> {  
    E myField;  
    class InnerClass<E> {  
        ...  
        E data = myField;  
    } // Not the same type!!  
}
```



# Generic Methods

---

- **A method can be generic when the class is not**

```
public static <E> void insertionSort  
(E[] array, Comparator<E> comparator);
```

- Define the type variable at the method

- **More generics**

<http://docs.oracle.com/javase/tutorial/java/generics/index.html>

# Wildcard

---

- **Used to denote super/subtype of type parameter**
- **Upper bounded wildcard:** `<? extends E>`
  - E and every subtype (subclass) of E
- **Lower bounded wildcard:** `<? super E>`
  - E and every supertype (superclass) of E
- **Consider** `<? extends E>` **for parameters,**  
`<? super E>` **for return type**
  - The only use in this project is with comparator

```
public BinarySearchTree(Comparator<? super E> c);
```

---

# Inheritance

Superclass & Interface

# Interface & Inheritance

---

- **Interface provides list of methods a class promise to implement**
  - Inheritance: is-a relationship *and* code sharing.
    - `AVLTree` can be treated as `BinarySearchTree` and inherits code.
  - Interfaces: is-a relationship *without* code sharing.
    - `FourHeap` can be treated as `PriorityQueue` but inherits no code.
- **Inheritance provides code reuse** **Style Points!!**
  - Take advantage of inherited methods
  - Do not re-implement already provided functionality
  - Override only when it is necessary

---

# Comparing Objects

Comparable & Comparator

# Comparing objects

---

- Operators  $<$ ,  $>$  do not work with objects in Java
- **Two ways of comparing:**
  1. Implement Comparable Interface
    - Natural Ordering: 1, 2, 3, 4 ...
    - One way of ordering
  2. Use Comparator **<- Project 2**
    - Many ways of ordering

# The Comparable interface

---

```
public interface Comparable<T> {  
    public int compareTo(T other);  
}
```

- A call of **A.compareTo(B)** should return:
  - a value  $<0$  if **A** comes "before" **B** in the ordering,
  - a value  $>0$  if **A** comes "after" **B** in the ordering,
  - or exactly  $0$  if **A** and **B** are considered "equal" in the ordering.

# What's the "natural" order?

---

```
public class Rectangle implements Comparable<Rectangle> {  
    private int x, y, width, height;  
  
    public int compareTo(Rectangle other) {  
        // ...?  
    }  
}
```

- What is the "natural ordering" of rectangles?
  - By x, breaking ties by y?
  - By width, breaking ties by height?
  - By area? By perimeter?
- Do rectangles have any "natural" ordering?
  - Might we ever want to sort rectangles into some order anyway?



# Comparator interface

---

```
public interface Comparator<T> {  
    public int compare(T first, T second);  
}
```

- **Interface Comparator:**
  - External object specifies comparison function
  - Can define multiple orderings

# Comparator examples

---

```
public class RectangleAreaComparator
    implements Comparator<Rectangle> {
    // compare in ascending order by area (WxH)
    public int compare(Rectangle r1, Rectangle r2) {
        return r1.getArea() - r2.getArea();
    }
}
```

```
public class RectangleXYComparator
    implements Comparator<Rectangle> {
    // compare by ascending x, break ties by y
    public int compare(Rectangle r1, Rectangle r2) {
        if (r1.getX() != r2.getX()) {
            return r1.getX() - r2.getX();
        } else {
            return r1.getY() - r2.getY();
        }
    }
}
```

# Using Comparators

---

- TreeSet and TreeMap can accept a Comparator parameter.

```
Comparator<Rectangle> comp = new RectangleAreaComparator();  
Set<Rectangle> set = new TreeSet<Rectangle>(comp);
```

- Searching and sorting methods can accept Comparators.

```
Arrays.binarySearch(array, value, comparator)  
Arrays.sort(array, comparator)  
Collections.binarySearch(list, comparator)  
Collections.max(collection, comparator)  
Collections.min(collection, comparator)  
Collections.sort(list, comparator)
```

- Methods are provided to reverse a Comparator's ordering:

```
Collections.reverseOrder()  
Collections.reverseOrder(comparator)
```

---

# Iterator

*objects that traverse collections*

# Iterator

---

- Object that allows traverse elements of collection
  - Anonymous class: Combined class declaration and instantiation.

```
public SimpleIterator<DataCount<E>> getIterator() {  
  
    return new SimpleIterator<DataCount<E>>() {  
        // Returns true if there are more elements to examine  
        public boolean hasNext() {  
            ...  
        }  
        // Returns the next element from the collection  
        public DataCount<E> next() {  
            if(!hasNext()) {  
                throw new NoSuchElementException();  
            }  
            ...  
        }  
    };  
}
```

---

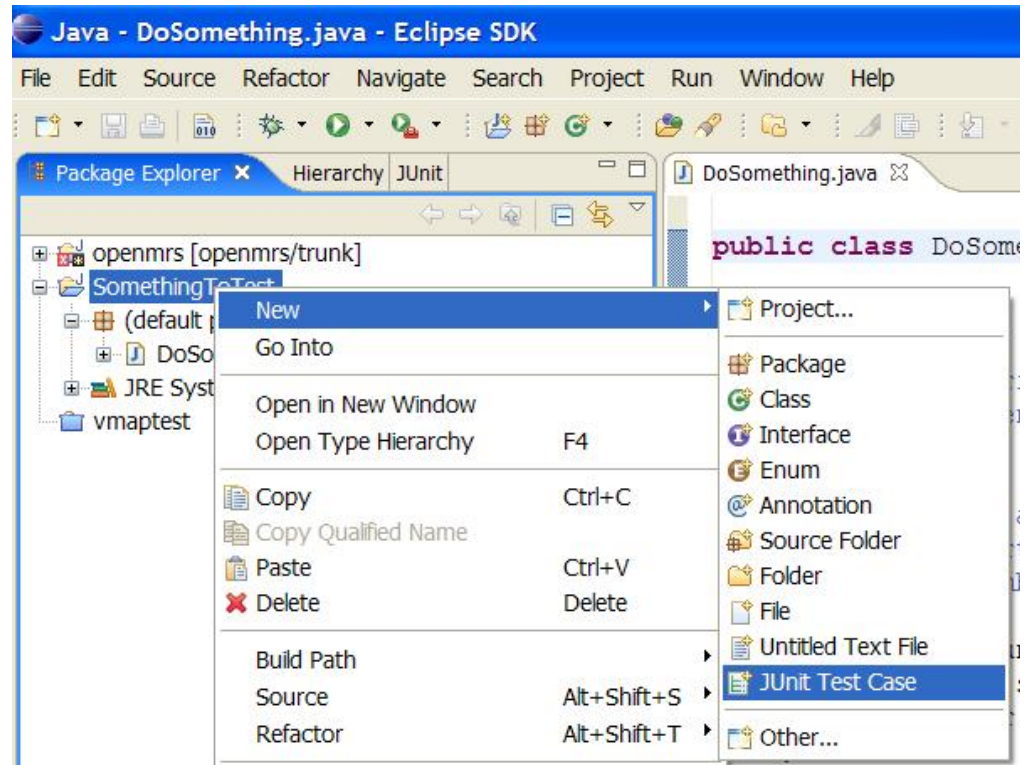
# JUnit

*Unit testing: Looking for errors in a subsystem in isolation*

# JUnit and Eclipse

- To add JUnit to an Eclipse project, click:
  - **Project** → **Properties** → **Build Path** → **Libraries** → **Add Library...** → **JUnit** → **JUnit 4** → **Finish**

- To create a test case:
  - right-click a file and choose **New** → **Test Case**
  - or click **File** → **New** → **JUnit Test Case**
  - Eclipse can create stubs of method tests for you.



# A JUnit test class

---

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

public class name {
    ...

    @Test
    public void name() { // a test case method
        ...
    }
}
```

- A method with `@Test` is flagged as a JUnit test case.
  - All `@Test` methods run when JUnit runs your test class.



# JUnit assertion methods

---

<code>assertTrue (<b>test</b>)</code>	fails if the boolean test is <code>false</code>
<code>assertFalse (<b>test</b>)</code>	fails if the boolean test is <code>true</code>
<code>assertEquals (<b>expected</b>, <b>actual</b>)</code>	fails if the values are not equal
<code>assertSame (<b>expected</b>, <b>actual</b>)</code>	fails if the values are not the same (by <code>==</code> )
<code>assertNotSame (<b>expected</b>, <b>actual</b>)</code>	fails if the values <i>are</i> the same (by <code>==</code> )
<code>assertNotNull (<b>value</b>)</code>	fails if the given value is <i>not</i> <code>null</code>
<code>assertNotNull (<b>value</b>)</code>	fails if the given value is <code>null</code>
<code>fail ()</code>	causes current test to immediately fail

- Each method can also be passed a string to display if it fails:
  - e.g. `assertEquals ("message", expected, actual)`
  - Why is there no `pass` method?

# Trustworthy tests

---

- Test **one thing at a time** per test method.
  - 10 small tests are much better than 1 test 10x as large.
- Each test method should have few (likely 1) assert statements.
  - If you assert many things, the first that fails stops the test.
  - You won't know whether a later assertion would have failed.
- Tests should avoid logic.
  - minimize `if/else`, `loops`, `switch`, etc.
- Torture tests are okay, but only *in addition to* simple tests.

# Naming test cases

---

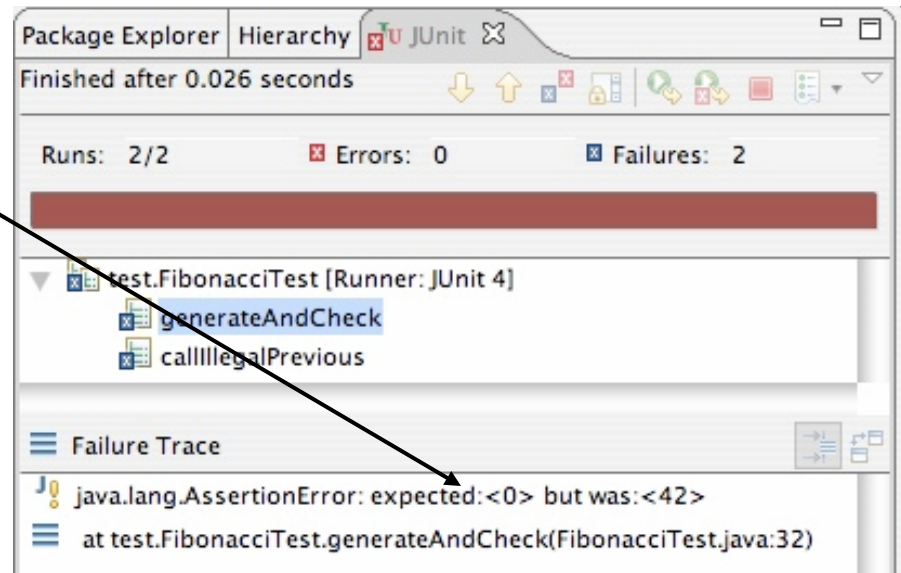
```
public class DateTest {
    @Test
    public void test_addDays_withinSameMonth_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals("date after +4 days", expected, actual);
    }
    // give test case methods really long descriptive names

    @Test
    public void test_addDays_wrapToNextMonth_2() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, actual);
    }
    // give descriptive names to expected/actual values
}
```

# Good assertion messages

```
public class DateTest {
    @Test
    public void test_addDays_addJustOneDay_1() {
        Date actual = new Date(2050, 2, 15);
        actual.addDays(1);
        Date expected = new Date(2050, 2, 16);
        assertEquals("adding one day to 2050/2/15",
            expected, actual);
    }
    ...
}
```

```
// JUnit will already show
// the expected and actual
// values in its output;
//
// don't need to repeat them
// in the assertion message
```



# Well-structured assertions

---

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        assertEquals(2050, d.getYear()); // expected
        assertEquals(2, d.getMonth()); // value should
        assertEquals(19, d.getDay()); // be at LEFT
    }

    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        assertEquals("year after +14 days", 2050, d.getYear());
        assertEquals("month after +14 days", 3, d.getMonth());
        assertEquals("day after +14 days", 1, d.getDay());
    } // test cases should usually have messages explaining
    // what is being checked, for better failure output
}
```

# Expected answer objects

---

```
public class DateTest {
    @Test
    public void test1() {
        Date d = new Date(2050, 2, 15);
        d.addDays(4);
        Date expected = new Date(2050, 2, 19);
        assertEquals(expected, d); // use an expected answer
                                   // object to minimize tests
    }

                                   // (Date must have toString
                                   // and equals methods)
    @Test
    public void test2() {
        Date d = new Date(2050, 2, 15);
        d.addDays(14);
        Date expected = new Date(2050, 3, 1);
        assertEquals("date after +14 days", expected, d);
    }
}
```

# Tests with a timeout

---

```
@Test(timeout = 5000)
public void name() { ... }
```

- The above method will be considered a failure if it doesn't finish running within 5000 ms

```
private static final int TIMEOUT = 2000;
...
```

```
@Test(timeout = TIMEOUT)
public void name() { ... }
```

- Times out / fails after 2000 ms

# Testing for exceptions

---

```
@Test(expected = ExceptionType.class)  
public void name() {  
    ...  
}
```

- Will pass if it *does* throw the given exception.
  - If the exception is *not* thrown, the test fails.
  - Use this to test for expected errors.

```
@Test(expected = ArrayIndexOutOfBoundsException.class)  
public void testBadIndex() {  
    ArrayList list = new ArrayList();  
    list.get(4);    // should fail  
}
```



# Setup and teardown

---

## @Before

```
public void name() { ... }
```

## @After

```
public void name() { ... }
```

- methods to run before/after each test case method is called

## @BeforeClass

```
public static void name() { ... }
```

## @AfterClass

```
public static void name() { ... }
```

- methods to run once before/after the entire test class runs

# Flexible helpers

---

```
public class DateTest {
    @Test(timeout = DEFAULT_TIMEOUT)
    public void addDays_multipleCalls_wrapToNextMonth2x() {
        Date d = addHelper(2050, 2, 15, +14, 2050, 3, 1);
        addhelper(d, +32, 2050, 4, 2);
        addhelper(d, +98, 2050, 7, 9);
    }

    // Helpers can box you in; hard to test many calls/combine.
    // Create variations that allow better flexibility
    private Date addHelper(int y1, int m1, int d1, int add,
                            int y2, int m2, int d2) {
        Date date = new Date(y, m, d);
        addHelper(date, add, y2, m2, d2);
        return d;
    }

    private void addHelper(Date date, int add,
                            int y2, int m2, int d2) {
        date.addDays(add);
        Date expect = new Date(y2, m2, d2);
        assertEquals("date after +" + add + " days", expect,
d);
    }
}
```

# Test case "smells"

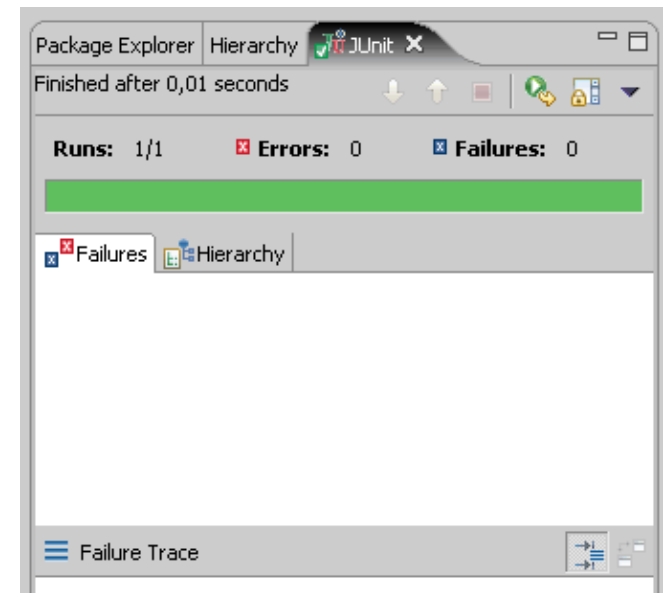
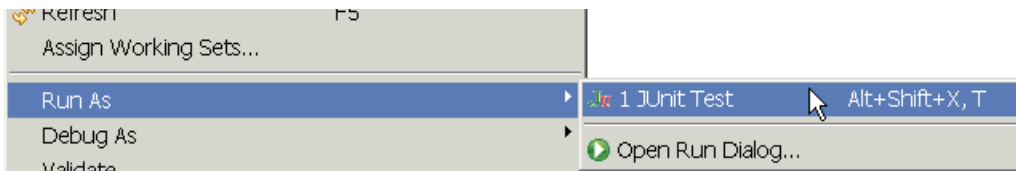
---

- Tests should be self-contained and not care about each other.
- "Smells" (bad things to avoid) in tests:
  - *Constrained test order* : Test A must run before Test B.  
(usually a misguided attempt to test order/flow)
  - *Tests call each other* : Test A calls Test B's method  
(calling a shared helper is OK, though)
  - *Mutable shared state* : Tests A/B both use a shared object.  
(If A breaks it, what happens to B?)



# Running a test

- Right click it in the Eclipse Package Explorer at left; choose:  
**Run As → JUnit Test**
- The JUnit bar will show **green** if all tests pass, **red** if any fail.
- The Failure Trace shows which tests failed, if any, and why.



---

# JGrasp Debugger

*Visual Tool for tree/list debugging*

# JGrasp Debugger

The screenshot displays the JGrasp Debugger's heap viewer interface. The window title is "Viewer (by name): heap". The main area shows a data array and a binary tree structure.

**Viewer (by name): heap**

heap

Type: FourHeap [Current] | Viewer: Presentation - Structu...

Width: 4.0 | Scale: .66

Animation Time: 1.0sec.

data: [ 1 | 3 | 6 | 5 | 8 | 9 | 14 | 13 | . | . ]

size: 8

The binary tree structure is as follows:

```
graph TD; 1((1)) --- 3((3)); 1 --- 6((6)); 3 --- 5((5)); 3 --- 8((8)); 6 --- 9((9)); 6 --- 14((14)); 5 --- 13((13));
```

The tree is a binary tree with root node 1. Node 1 has children 3 and 6. Node 3 has children 5 and 8. Node 6 has children 9 and 14. Node 5 has child 13. The nodes are represented by circles with their values inside. The root node 1 is highlighted with a red border. The leaf nodes 5, 8, 9, 14, and 13 are highlighted with a green border. The nodes 3 and 6 are highlighted with a blue border. The data array and size field are also highlighted with a yellow border.

---

# **Project 1 Feedback & Project 2 Tips**

*Style Guide*

# Style Guide

---

- **Style Points are 1/3 of your grade!!**
  - Grade breakdown: 1/3 correctness, 1/3 write up, 1/3 style
  - In industry, you'll be always working with other people
  - Readability and Conciseness of code is very very important!!!
- **Make sure you read style guide**
  - Style guide: <http://www.cs.washington.edu/education/courses/cse332/13wi/projects/style.txt>
  - Comment guide: <http://www.cs.washington.edu/education/courses/cse332/13wi/projects/commenting.pdf>
  - Java Convention: <http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>



# Project 1 Feedback

---

- **Make sure your code compile!!**
  - No correctness point if your code doesn't compile
  - Be sure to take out package statement
- **Comment your code**
  - Follow guide: <http://www.cs.washington.edu/education/courses/cse332/13wi/projects/style.txt>
- **Use descriptive variable / method names**
  - If variable points to stack top, name it like 'stackTop', not 'l' or 'g'
- **Use specify private / public appropriately**
  - Do not just omit this!

# Project 1 Feedback

---

- **Initialize all non-static fields in constructor**

```
private int size = 0; // 😞
```

```
private int size; Public Stack() { size = 0; } // 😊
```

- **Make your code as concise as possible**

```
end = end + 1; // 😞 😞 😞
```

```
this.isEmpty();
```

```
front = this.new Node(d, front);
```

```
if(front == null){  
    front = new ListNode(d);  
}else{  
    ListNode newNode = new ListNode(d, front);  
    front = newNode;  
}
```

```
end ++; // 😊 😊 😊
```

```
isEmpty();
```

```
front = new Node(d, front);
```

```
front = new ListNode(d, front);
```

# Project 1 Feedback

---

- **Use constants for fixed constants**

```
private static final int INITIAL_CAPACITY = 10;
private static final int RESIZE_FACTOR = 2;
```

- **Use Boolean zen**

```
if(size==0){ return true; }else{ return false; } // 😞
return size == 0; // 😊
```

- **Maximize code reuse, minimize redundancy**

- For example, calling `peek()` inside of `pop()`
- Using `isEmpty()` instead of `size == 0` // **Better Readability**

# Project 1 Feedback

---

- **Use `@Override` when overriding**
- **Do not leave warning generating code**
  - Unless you know why it is there and why it is unavoidable (Generic array, casting for AVLNode)
  - Suppress warnings on method/variable, but not on whole class
- **Avoid inefficient design**
  - ArrayStack: Insert into `stackArray[0]`, shifting all elements
- **Avoid unnecessary field**

```
private int size = array.length;
```

# Project 2 Tips

---

- Take advantage of superclass's implementation when writing subclass
- Minimize casting
  - Remember **AVLNode is-a BSTNode**
  - AVLNode can be treated as BSTNode, **only except** when accessing its **height** information
  - Consider some private function like (**only cast in this function**):  
`int height(BSTNode node)`  
`void updateHeight(BSTNode node)`