Plan for Today

- Web-CAT Announcements
- Review terminology from last class
- Finish Recipe implementation of StackInt as a class
- Abstraction Barrier and Interchangeable Parts
- Cover Liskov Chapters 1 and 2
- Discuss Assignment 1
- Start discussing Liskov Chapter 3
Abstract Data Type (ADT)

• What is an ADT?
  - set of data
  - set of operations
  - description of what operations do

• Within this course, when discuss ADTs, we will discuss them using:
  - a signature: names of operations and types
  - a specification: agreement between client and implementors
StackInt Signature

empty:                    -->  StackInt
push:     StackInt x int  -->  StackInt
isEmpty:  StackInt        -->  boolean
top:      StackInt        -->  int
pop:      StackInt        -->  StackInt
size:     StackInt        -->  int
StackInt

Algebraic Specifications

isEmpty (empty()) = true
isEmpty (push (s, n)) = false

top (push (s, n)) = n

pop (push (s, n)) = s

size (empty()) = 0
size (push (s, n)) = 1 + size (s)
Recipe Implementation of StackInt
StackInt Recipe Implementation

Determine which operations of the ADT are basic creators, and which are other operations.

Hints:

- Except for the basic creators, each operation is specified by one or more equations.

- The basic creators are used as arguments in the left side of the equations.

empty: \( \rightarrow \) StackInt
push: \( \text{StackInt} \times \text{int} \rightarrow \text{StackInt} \)
isEmpty: \( \text{StackInt} \rightarrow \text{boolean} \)
top: \( \text{StackInt} \rightarrow \text{int} \)
pop: \( \text{StackInt} \rightarrow \text{StackInt} \)
size: \( \text{StackInt} \rightarrow \text{int} \)

isEmpty (empty()) = true
isEmpty (push (s, n)) = false
top (push (s, n)) = n
pop (push (s, n)) = s
size (empty()) = 0
size (push (s, n)) = 1 + size (s)
StackInt Recipe Implementation

Determine which operations of the ADT are basic creators, and which are other operations.

Hints:
- Except for the basic creators, each operation is specified by one or more equations.
- The basic creators are used as arguments in the left side of the equations.

The basic creators are empty and push.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Signature</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>empty</td>
<td>StackInt</td>
<td>-&gt; StackInt</td>
</tr>
<tr>
<td>push</td>
<td>StackInt x int</td>
<td>-&gt; StackInt</td>
</tr>
<tr>
<td>isEmpty</td>
<td>StackInt</td>
<td>-&gt; boolean</td>
</tr>
<tr>
<td>top</td>
<td>StackInt</td>
<td>-&gt; int</td>
</tr>
<tr>
<td>pop</td>
<td>StackInt</td>
<td>-&gt; StackInt</td>
</tr>
<tr>
<td>size</td>
<td>StackInt</td>
<td>-&gt; int</td>
</tr>
</tbody>
</table>

isEmpty (empty()) = true
isEmpty (push(s, n)) = false
top (push(s, n)) = n
pop (push(s, n)) = s
size (empty()) = 0
size (push(s, n)) = 1 + size (s)
StackInt Recipe Implementation

Define an abstract class named T.
StackInt Recipe Implementation

Define an abstract class named T.

```java
/**
 * StackInt represents a stack
 * of ints
 */

public abstract class StackInt {
}
```
StackInt Recipe Implementation

For each basic creator c of the ADT, define a concrete subclass of T.
StackInt Recipe Implementation

For each basic creator c of the ADT, define a concrete subclass of T.

```java
/**
 * Empty represents a stack that contains no ints
 */
class Empty extends StackInt {}

/**
 * Push represents a non-empty stack of ints
 */
class Push extends StackInt {}
```
StackInt Recipe Implementation

For each concrete subclass, declare instance variables with the same types and names as the arguments that are passed to the corresponding basic creator c.

empty: -> StackInt
push: StackInt x int -> StackInt
StackInt Recipe Implementation

For each concrete subclass, declare instance variables with the same types and names as the arguments that are passed to the corresponding basic creator c.

```java
empty:                -> StackInt
push: StackInt x int -> StackInt

/**
 * Empty represents a stack that
 * contains no ints
 */
class Empty extends StackInt {}

/**
 * Push represents a non-empty
 * stack of ints
 */
class Push extends StackInt {
  /** other elements of this StackInt */
  StackInt s;
  /** topmost element of this StackInt */
  int n;
}
```
StackInt Recipe Implementation

For each concrete subclass, define a Java constructor that takes the same arguments as the basic creator `c` and stores those arguments into the instance variables.

empty: -> StackInt
push: StackInt x int -> StackInt
StackInt Recipe Implementation

For each concrete subclass, define a Java constructor that takes the same arguments as the basic creator $c$ and stores those arguments into the instance variables.

empty: $\rightarrow$ StackInt

push: StackInt $\times$ int $\rightarrow$ StackInt

/**
 * Constructor for Empty
 */
Empty () { }

/**
 * Constructor for Push
 * @param s the stack without topmost element
 * @param n is the topmost element of the stack
 */
Push (StackInt s, int n) {
    this.s = s;
    this.n = n;
}
StackInt Recipe Implementation

For each operation \( f \) of the ADT that is not a basic creator, declare an abstract method \( f \) in the abstract class \( T \).

The explicit arguments of the abstract method \( f \) should be the same as the arguments of the operation \( f \), except the abstract method \( f \) will take one less argument than \( f \). That missing argument will be the receiver for calls to the abstract method \( f \) (so that argument will still be available to the abstract method \( f \), as the value of Java's special variable \( this \)). That missing argument should be of type \( T \). If the operation \( f \) has more than one argument of type \( T \), so you have some choice as to which argument to omit, then you should omit the argument that discriminates between the various equations for \( f \) in the algebraic specification.

\[
\begin{align*}
\text{empty:} & \quad \rightarrow \text{StackInt} \\
\text{push:} & \quad \text{StackInt} \times \text{int} \rightarrow \text{StackInt} \\
\text{isEmpty:} & \quad \text{StackInt} \rightarrow \text{boolean} \\
\text{top:} & \quad \text{StackInt} \rightarrow \text{int} \\
\text{pop:} & \quad \text{StackInt} \rightarrow \text{StackInt} \\
\text{size:} & \quad \text{StackInt} \rightarrow \text{int}
\end{align*}
\]

\[
\begin{align*}
\text{isEmpty} (\text{empty}()) & = \text{true} \\
\text{isEmpty} (\text{push} (s, n)) & = \text{false} \\
\text{top} (\text{push} (s, n)) & = n \\
\text{pop} (\text{push} (s, n)) & = s \\
\text{size} (\text{empty}()) & = 0 \\
\text{size} (\text{push} (s, n)) & = 1 + \text{size} (s)
\end{align*}
\]
StackInt Recipe Implementation

For each operation \( f \) of the ADT that is not a basic creator, declare an abstract method \( f \) in the abstract class \( T \).

The explicit arguments of the abstract method \( f \) should be the same as the arguments of the operation \( f \), except the abstract method \( f \) will take one less argument than \( f \). That missing argument will be the receiver for calls to the abstract method \( f \) (so that argument will still be available to the abstract method \( f \), as the value of Java's special variable this). That missing argument should be of type \( T \). If the operation \( f \) has more than one argument of type \( T \), so you have some choice as to which argument to omit, then you should omit the argument that discriminates between the various equations for \( f \) in the algebraic specification.

\[
\begin{align*}
\text{empty:} & \quad \rightarrow \text{StackInt} \\
\text{push:} & \quad \text{StackInt} \times \text{int} \rightarrow \text{StackInt} \\
\text{isEmpty:} & \quad \text{StackInt} \rightarrow \text{boolean} \\
\text{top:} & \quad \text{StackInt} \rightarrow \text{int} \\
\text{pop:} & \quad \text{StackInt} \rightarrow \text{StackInt} \\
\text{size:} & \quad \text{StackInt} \rightarrow \text{int}
\end{align*}
\]

\[
\begin{align*}
\text{isEmpty} (\text{empty}()) & = \text{true} \\
\text{isEmpty} (\text{push} (s, n)) & = \text{false} \\
\text{top} (\text{push} (s, n)) & = n \\
\text{pop} (\text{push} (s, n)) & = s \\
\text{size} (\text{empty}()) & = 0 \\
\text{size} (\text{push} (s, n)) & = 1 + \text{size} (s)
\end{align*}
\]
StackInt Recipe Implementation

For each operation f of the ADT that is not a basic creator, declare an abstract method f in the abstract class T.

The explicit arguments of the abstract method f should be the same as the arguments of the operation f, except the abstract method f will take one less argument than f. That missing argument will be the receiver for calls to the abstract method f (so that argument will still be available to the abstract method f, as the value of Java's special variable this). That missing argument should be of type T. If the operation f has more than one argument of type T, so you have some choice as to which argument to omit, then you should omit the argument that discriminates between the various equations for f in the algebraic specification.

empty:                  -> StackInt
push:    StackInt x int -> StackInt
isEmpty: StackInt       -> boolean
top:     StackInt       -> int
pop:     StackInt       -> StackInt
size:    StackInt       -> int

abstract boolean isEmptyMethod ();
abstract int topMethod ();
abstract StackInt popMethod ();
abstract int sizeMethod ();
StackInt Recipe Implementation

For each operation of the ADT, define a static method within the abstract class.

- If the operation is a basic creator c, then the static method for c should create and return a new instance of the subclass that corresponds to c.

- If the operation is not a basic creator, then the static method for c should delegate to the corresponding abstract method for c, passing it all but one of its arguments. (The missing argument will be available to the abstract method via the special variable named “this”.) Suppose, for example, that the static method T.f is called with arguments x1, x2, x3, and x4, where x1 is the only argument of type T. Then the body of T.f should be: return x1.f (x2, x3, x4);

empty:                 -> StackInt
push:    StackInt x int -> StackInt
isEmpty: StackInt       -> boolean
top:     StackInt       -> int
pop:     StackInt       -> StackInt
size:    StackInt       -> int
For each operation of the ADT, define a static method within the abstract class.

- If the operation is a basic creator c, then the static method for c should create and return a new instance of the subclass that corresponds to c.

- If the operation is not a basic creator, then the static method for c should delegate to the corresponding abstract method for c, passing it all but one of its arguments. (The missing argument will be available to the abstract method via the special variable named “this”.) Suppose, for example, that the static method T.f is called with arguments x1, x2, x3, and x4, where x1 is the only argument of type T. Then the body of T.f should be:
  
  ```java
  return x1.f(x2, x3, x4);
  ```
StackInt Recipe Implementation

For each operation of the ADT, define a static method within the abstract class.

- If the operation is a basic creator $c$, then the static method for $c$ should create and return a new instance of the subclass that corresponds to $c$.

- If the operation is not a basic creator, then the static method for $c$ should delegate to the corresponding abstract method for $c$, passing it all but one of its arguments. (The missing argument will be available to the abstract method via the special variable named “this”.) Suppose, for example, that the static method $T.f$ is called with arguments $x_1$, $x_2$, $x_3$, and $x_4$, where $x_1$ is the only argument of type $T$. Then the body of $T.f$ should be:

```
public static StackInt empty () {
    return new Empty ();
}
```

```
public static StackInt push (StackInt s, int n) {
    return new Push (s, n);
}
```
StackInt Recipe Implementation

For each operation of the ADT, define a static method within the abstract class.

- If the operation is a basic creator c, then the static method for c should create and return a new instance of the subclass that corresponds to c.

- If the operation is not a basic creator, then the static method for c should delegate to the corresponding abstract method for c, passing it all but one of its arguments. (The missing argument will be available to the abstract method via the special variable named “this”.) Suppose, for example, that the static method T.f is called with arguments x1, x2, x3, and x4, where x1 is the only argument of type T. Then the body of T.f should be:

```java
return x1.f (x2, x3, x4);
```
StackInt Recipe Implementation

For each operation of the ADT, define a static method within the abstract class.

- If the operation is a basic creator c, then the static method for c should create and return a new instance of the subclass that corresponds to c.

- If the operation is not a basic creator, then the static method for c should delegate to the corresponding abstract method for c, passing it all but one of its arguments. (The missing argument will be available to the abstract method via the special variable named “this”.) Suppose, for example, that the static method T.f is called with arguments x1, x2, x3, and x4, where x1 is the only argument of type T. Then the body of T.f should be:

```java
public static boolean isEmpty (StackInt s) {
    return s.isEmptyMethod();
}
```

```java
public static int top (StackInt s) {
    return s.topMethod();
}
```

```java
public static StackInt pop (StackInt s) {
    return s.popMethod();
}
```

```java
public static int size (StackInt s) {
    return s.sizeMethod();
}
```
StackInt Recipe Implementation

For each concrete subclass C of T, define all of the abstract methods that were declared within the abstract base class for T. (These definitions define dynamic (as opposed to static) methods.)

For each abstract method f that is defined within a subclass C, the body of f should return the value specified by the algebraic specification for the case in which Java’s special variable this will be an instance of the class C.

- If the algebraic specification contains only one equation that describes the result of an operation f applied to an instance of C, and that equation has no side conditions, then the body of the dynamic method f should return the value expressed by the right hand side of that equation.

- If the algebraic specification does not contain any equations that describe the result of an operation f applied to an instance of C, then the body of the dynamic method f should throw a RuntimeException such as an IllegalArgumentException.

- (Note that other conditions are listed in the recipe.)

```
empty:                  -> StackInt
push:    StackInt x int -> StackInt
isEmpty: StackInt       -> boolean
top:     StackInt       -> int
pop:     StackInt       -> StackInt
size:    StackInt       -> int

isEmpty (empty())  =  true
isEmpty (push (s, n))  =  false
top (push (s, n))  =  n
pop (push (s, n))  =  s
size (empty())  =  0
size (push (s, n))  =  1 + size (s)
```
StackInt Recipe Implementation

For each concrete subclass C of T, define all of the abstract methods that were declared within the abstract base class for T. (These definitions define dynamic (as opposed to static) methods.)

For each abstract method f that is defined within a subclass C, the body of f should return the value specified by the algebraic specification for the case in which Java’s special variable this will be an instance of the class C.

-If the algebraic specification contains only one equation that describes the result of an operation f applied to an instance of C, and that equation has no side conditions, then the body of the dynamic method f should return the value expressed by the right hand side of that equation.

-If the algebraic specification does not contain any equations that describe the result of an operation f applied to an instance of C, then the body of the dynamic method f should throw a RuntimeException such as an IllegalArgumentException.

-(Note that other conditions are listed in the recipe.)

```java
isEmpty (empty())  =  true
isEmpty (push (s, n))  =  false
top (push (s, n))  =  n
pop (push (s, n))  =  s
size (empty())  =  0
size (push (s, n))  =  1 + size (s)

class Empty extends StackInt {

    Empty () { }

    boolean isEmptyMethod () {
        return true;
    }

    int topMethod () {
        String msg1 = "attempted to compute the top of an empty "
                       + "StackInt";
        throw new RuntimeException (msg1);
    }

    StackInt popMethod () {
        String msg1 = "attempted to pop from an empty StackInt";
        throw new RuntimeException (msg1);
    }

    int sizeMethod () {
        return 0;
    }
}
```
StackInt Recipe Implementation

For each concrete subclass C of T, define all of the abstract methods that were declared within the abstract base class for T. (These definitions define dynamic (as opposed to static) methods.)

For each abstract method f that is defined within a subclass C, the body of f should return the value specified by the algebraic specification for the case in which Java’s special variable this will be an instance of the class C.

-If the algebraic specification contains only one equation that describes the result of an operation f applied to an instance of C, and that equation has no side conditions, then the body of the dynamic method f should return the value expressed by the right hand side of that equation.

-If the algebraic specification does not contain any equations that describe the result of an operation f applied to an instance of C, then the body of the dynamic method f should throw a RuntimeException such as an IllegalArgumentException.

- (Note that other conditions are listed in the recipe.)

```java
isEmpty (empty()) = true
isEmpty (push (s, n)) = false
top (push (s, n)) = n
pop (push (s, n)) = s
size (empty()) = 0
size (push (s, n)) = 1 + size (s)

class Push extends StackInt{
    StackInt s;
    int n;

    Push(StackInt s, int n){
        this.s = s;
        this.n = n;
    }

    boolean isEmptyMethod(){
        return false;
    }

    int topMethod(){
        return n;
    }

    StackInt popMethod(){
        return s;
    }

    int sizeMethod(){
        return 1 + size(s);
    }
}
```
Abstract Class vs. Concrete Class

- Concrete class: full implementation of the type.
- Abstract classes: at most a partial implementation of the type
Abstraction Barrier

• Every piece of software has, or should have, an abstraction barrier that divides the world into two parts: clients and implementors.

- The clients are those who use the software. They do not need to know how the software works.

- The implementors are those who build it. They need to know how the software works.
Interchangeable Parts
[history.com]

• 18th century: Guns were made by hand, which meant each gun was one-of-a-kind. To repair a gun, a part had to be made especially for it.

• Mid-18th century a French gunsmith Honoré LeBlanc suggested the gun parts be made from standardized patterns, so that all gun parts would follow the same design and could be easily replaced if broken.

• Early 19th century, Eli Whitney manufacturing muskets—production of large numbers of identical parts quickly and at a relatively low cost.

• The U.S. introduced the first large-scale assembly of weapons with its adoption of the Model 1842 musket, and the new arms industry would produce hundreds of thousands of rifles for Civil War soldiers, all from interchangeable parts.
Liskov Chapter 2
Type Checking

• Type safety

• Strongly type checking
Type Hierarchy
[Liskov, p.27, Sidebar 2.4]

• Java supports type hierarchy, in which one type can be the supertype of other types, which are its subtypes. A subtype’s objects have all the methods defined by the supertype.

• All object types are subtypes of Object, which is the top of the type hierarchy. Object defines a number of methods, including equals and toString. Every object is guaranteed to have these methods.

• The apparent type of a variable is the type understood by the compiler from information available in declarations. The actual type of an object is its real type—the type it receives when it is created.

• Java guarantees that the apparent type of an expression is a super type of its actual type.
Abstraction
Assignment 1

• Due: Tuesday, September 17, 2013 at 11:59 pm

• Recipe implementation in Java of the FSetString ADT

• FSetString is an immutable abstract data type whose values represent finite sets with elements of type String.
Recipe Assumption

Except for the basic creators, each operation is specified by one or more equations. If an operation is specified by more than one equation, then the left hand sides of the equations differ according to which basic creator was used to create an argument of type $T$. 

Recipe Assumption

Except for the basic creators, each operation is specified by one or more equations. If an operation is specified by more than one equation, then the left hand sides of the equations differ according to which basic creator was used to create an argument of type T.

\[
\text{FSetString.add}(s_0, k_0) = s_0 \\
\quad \text{if } \text{FSetString.contains}(s_0, k_0) \\
\text{FSetString.add}(s_0, k_0) = \text{FSetString.insert}(s_0, k_0) \\
\quad \text{if } !\left(\text{FSetString.contains}(s_0, k_0)\right)
\]
Recipe Assumption

The operations of the ADT are to be implemented as static methods of a class named T.
equals method

• If s1 is a value of the FSetString ADT, then s1.equals(null) returns false.

• If s1 is a value of the FSetString ADT, but x is not, then s1.equals(x) returns false.

• If s1 and s2 are values of the FSetString ADT, then s1.equals(s2) if and only if for every String k

  FSetString.contains(s1, k) if and only if FSetString.contains(s2, k)
Theorem

The following conditions are equivalent:

1. for every String k, \( \text{FSetString.contains}(s_1, k) \) if and only if \( \text{FSetString.contains}(s_2, k) \)

2. \( \text{FSetString.isSubset}(s_1, s_2) \) and \( \text{FSetString.isSubset}(s_2, s_1) \)
hashCode method

If s1 and s2 are values of the FSetString ADT, and 
s1.equals(s2), then s1.hashCode() == s2.hashCode().