Plan for Today

• Assignment 2
• Abc test cases
• Data Abstraction
• In-Class Exercise
Signature:

Public static methods (of the Abc class):

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>defg</td>
<td>Abc x int</td>
<td>int</td>
</tr>
<tr>
<td>hijk</td>
<td>Abc x int</td>
<td>Abc</td>
</tr>
<tr>
<td>lmno</td>
<td>Abc x int</td>
<td>Abc</td>
</tr>
<tr>
<td>pqrs</td>
<td>int</td>
<td>Abc</td>
</tr>
<tr>
<td>tuvw</td>
<td>Abc</td>
<td>int</td>
</tr>
</tbody>
</table>

Algebraic Specification:

\[
\begin{align*}
Abc.\text{defg} (\text{Abc.lmno} (u, k), n) &= \text{defg} (u, n) & \text{if } n < \text{tuvw} (u) \\
Abc.\text{defg} (\text{Abc.lmno} (u, k), n) &= k & \text{if } n == \text{tuvw} (u) \\
Abc.\text{defg} (\text{Abc.lmno} (u, k), n) &= n & \text{if } n > \text{tuvw} (u) \\
Abc.\text{defg} (\text{Abc.pqrs} (k), n) &= 3 \\
Abc.\text{hijk} (\text{Abc.lmno} (u, k), n) &= \text{lmno} (\text{hijk} (u, n), k) & \text{if } n < \text{tuvw} (u) \\
Abc.\text{hijk} (\text{Abc.lmno} (u, k), n) &= \text{lmno} (u, n + 1) & \text{if } n == \text{tuvw} (u) \\
Abc.\text{hijk} (\text{Abc.lmno} (u, k), n) &= u & \text{if } n > \text{tuvw} (u) \\
Abc.\text{hijk} (\text{Abc.pqrs} (k), n) &= \text{lmno} (\text{pqrs} (0), k) \\
Abc.\text{tuvw} (\text{Abc.lmno} (u, k)) &= 1 + \text{tuvw} (u) \\
Abc.\text{tuvw} (\text{Abc.pqrs} (k)) &= 0
\end{align*}
\]
Abc Test Cases

f1 = Abc.pqrs(1); //1
f2 = Abc.lmno (f1, 2); //1,2
f3 = Abc.lmno (f2, 3); //1,2,3
f4 = Abc.lmno (f3, 4); //1,2,3,4

assertTrue("tuvw f1", Abc.tuvw(f1)==0);
assertTrue("tuvw f2", Abc.tuvw(f2)==1);
assertTrue("tuvw f3", Abc.tuvw(f3)==2);
assertTrue("tuvw f4", Abc.tuvw(f4)==3);
Abc Test Cases

\[
\begin{align*}
  f_1 &= \text{Abc.pqrs}(1); // 1 \\
  f_2 &= \text{Abc.lmno}(f_1, 2); // 1, 2 \\
  f_3 &= \text{Abc.lmno}(f_2, 3); // 1, 2, 3 \\
  f_4 &= \text{Abc.lmno}(f_3, 4); // 1, 2, 3, 4
\end{align*}
\]

\[
\begin{align*}
  \text{assertTrue}("\text{defg f}_1 \ 1", \text{Abc.defg}(f_1, 1) == 3); \\
  \text{assertTrue}("\text{defg f}_1 \ 2", \text{Abc.defg}(f_1, 2) == 3); \\
  \text{assertTrue}("\text{defg f}_4 \ 1", \text{Abc.defg}(f_4, 1) == 3); \\
  \text{assertTrue}("\text{defg f}_4 \ 2", \text{Abc.defg}(f_4, 2) == 4); \\
  \text{assertTrue}("\text{defg f}_4 \ 3", \text{Abc.defg}(f_4, 3) == 3); \\
  \text{assertTrue}("\text{defg f}_4 \ 4", \text{Abc.defg}(f_4, 4) == 4);
\end{align*}
\]
Abc Test Cases

f1 = Abc.pqrs(1); //1
f2 = Abc.lmno (f1, 2); //1,2
f3 = Abc.lmno (f2, 3); //1,2,3
f4 = Abc.lmno (f3, 4); //1,2,3,4

assertTrue("hijk f1, 4", Abc.hijk(f1, 4).equals(Abc.lmno(Abc.pqrs(0), 1)));
assertTrue("hijk f2, -2", Abc.hijk(f2,-2).equals(Abc.lmno(Abc.lmno(Abc.pqrs(0), 1), 2)));
assertTrue("hijk f1 1", Abc.hijk(f1,1).equals(Abc.lmno (Abc.pqrs (0), 1)));
assertTrue("hijk f4 1", Abc.hijk(f4,1).equals(Abc.lmno(Abc.lmno(f2,2),4)));
assertTrue("hijk f4 2", Abc.hijk(f4,2).equals(Abc.lmno (f3, 3)));
assertTrue("hijk f4 3", Abc.hijk(f4,3).equals(f3));
Abstraction Mechanisms

- Abstraction by parameterization
- Abstraction by specification
Kinds of Abstraction

• Procedural abstraction
• Data abstraction
• Iteration abstraction
What is data abstraction?
What is data abstraction?

A type of abstraction that allows us to introduce new types of data objects.
What must we define with a new data type?
What must we define with a new data type?

- set of objects
- set of operations characterizing the behavior of the objects

\[
data \text{ abstraction} = \langle \text{objects}, \text{operations}\rangle\]
Abstract Data Type (ADT) Review

• 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
Objects

- Object
  - a programming entity that contains state (data) and behavior (methods)

- Objects we’ve discussed so far...
  - String
  - Point
  - Scanner
  - Random
  - File
  - arrays
Objects

- **State**: a set of values (internal data) stored in an object

- **Behavior**: a set of actions an object can perform, often reporting or modifying its internal state
Client Code

- Objects themselves are not complete programs; they are components that are given distinct roles and responsibilities.

- Objects can be used as part of larger programs to solve programs.

- **Client (or Client Code):** code that interacts with a class or objects of that class.
What do we gain from data abstraction?
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.
Abstraction Barrier

- **Client**
  - Knows the behavior of the data type
  - Doesn’t know how the data type was implemented, but can use the data type based on the specs

- **Implementor**
  - Knows the behavior of the data type
  - Knows how the data type was implemented
Which abstraction mechanisms are used with data abstraction?
Which abstraction mechanisms are used with data abstraction?

- Abstraction by parameterization
- Abstraction by specification
Specifications

• Formal
• Informal
visibility class dname{
    //OVERVIEW: A brief description of the
    // behavior of the type’s objects goes
    // here.

    //constructors
    //specs for constructors go here

    //methods
    //specs for methods go here
}
public class IntSet{
    //OVERVIEW: IntSets are mutable, unbounded
    //          sets of integers.
    //          A typical IntSet is \{x_1, \ldots, x_n\}

    //constructors
    public IntSet()
        //EFFECTS: Initializes this to be empty

    //methods
    public void insert (int x)
        //MODIFIES: this
        //EFFECTS: Adds x to the elements of
        //         this, i.e.,
        //         this_post = this + \{x\}.

    public void remove (int x)
        //MODIFIES: this
        //EFFECTS: Removes x from this, i.e.,
        //         this_post = this - \{x\}

    public boolean isIn (int x)
        //EFFECTS: If x is in this returns true
        //         else returns false

    public int size ()
        //EFFECTS: Returns the cardinality of
        //         this

    public int choose () throws Empty Exception
        //EFFECTS: If this is empty, throws
        //         EmptyException else
        //         returns an arbitrary element of this
}
public class IntSet{
    //OVERVIEW: IntSets are mutable, unbounded
    //  sets of integers.
    //  A typical IntSet is \{x_1, ..., x_n\}

    //constructors
    public IntSet()
        //EFFECTS: Initializes this to be empty

    //methods
    public void insert (int x)
        //MODIFIES: this
        //EFFECTS: Adds x to the elements of this, i.e.,
        //  this_post = this + \{x\}.
    public void remove (int x)
        //MODIFIES: this
        //EFFECTS: Removes x from this, i.e.,
        //  this_post = this - \{x\}
    public boolean isIn (int x)
        //EFFECTS: If x is in this returns true
        //else returns false
    public int size ()
        //EFFECTS: Returns the cardinality of this
    public int choose () throws EmptyException
        //EFFECTS: If this is empty, throws EmptyException else
        //  returns an arbitrary element of this
}

emptySet : -> FSetString
insert : FSetString x String -> FSetString
add : FSetString x String -> FSetString
size : FSetString -> int
isEmpty : FSetString -> boolean
contains : FSetString x String -> boolean
absent : FSetString x String -> FSetString

FSetString.add(s0, k0) = s0
    if FSetString.contains(s0, k0)
FSetString.add(s0, k0) = FSetString.insert(s0, k0)
    if !(FSetString.contains(s0, k0))

FSetString.size(FSetString.emptySet()) = 0
FSetString.size(FSetString.insert(s0, k0))
    = FSetString.size(s0)
    if FSetString.contains(s0, k0)
FSetString.size(FSetString.insert(s0, k0))
    = 1 + FSetString.size(s0)
    if !(FSetString.contains(s0, k0))

FSetString.contains(FSetString.emptySet(), k) = false
FSetString.contains(FSetString.insert(s0, k0), k)
    = true
    if k.equals(k0)
FSetString.contains(FSetString.insert(s0, k0), k)
    = FSetString.contains(s0, k)
    if !k.equals(k0))

FSetString.absent(FSetString.emptySet(), k) =
FSetString.emptySet()
FSetString.absent(FSetString.insert(s0, k0), k)
    = FSetString.absent(s0, k)
    if k.equals(k0)
FSetString.absent(FSetString.insert(s0, k0), k)
    = FSetString.insert(FSetString.absent(s0, k0), k0)
    if !k.equals(k0))
public class Poly{
    // OVERVIEW: Polys are immutable polynomials with integer coefficients
    //  A typical Poly is c_0 + c_1 x + ...

    // constructors
    public Poly() {
        // EFFECTS: Initializes this to be the zero polynomial
    }
    public Poly(int c, int n) throws NegativeExponentException {
        // EFFECTS: If n<0 throws NegativeExponentException else
        // initializes this to be the Poly cx^n
    }

    // methods
    public int degree() {
        // EFFECTS: Returns the degree of this, i.e., the largest exponent
        // with a non-zero coefficient. Returns 0 if this is the zero Poly.
    }
    public int coeff(int d) {
        // EFFECTS: Returns the coefficient of the term of this whose exponent is d
    }
    public Poly add(Poly q) throws NullPointerException {
        // EFFECTS: If q is null throws NullPointerException else
        // returns the Poly this + q.
    }
    public Poly mul(Poly q) throws NullPointerException {
        // EFFECTS: If q is null throws NullPointerException else
        // returns the Poly this * q.
    }
    public Poly sub(Poly q) throws NullPointerException {
        // EFFECTS: If q is null throws NullPointerException else
        // returns the Poly this - q.
    }
    public Poly minus() {
        // EFFECTS: Returns the Poly - this.
    }
}
Implementing Data Abstractions
Access in Implementation
Access Modifiers

- **private** - accessible only within the same class
- **(default)** - accessible only within the same package
- **protected** - accessible within the same package and also accessible within subclasses
- **public** - accessible everywhere
Item 13: Minimize the accessibility of classes and members

[Bloch]
Item 45: Minimize the scope of local variables

[Bloch]
Item 14: In public classes, use accessor methods, not public fields

[Bloch]
Records
Sidebar 5.1 - equals, clone, and toString

[Liskov, p.94]

• Two objects are equals if they are behaviorally equivalent. Mutable objects are equals only if they are the same object; such types can inherit equals from Object. Immutable objects are equals if they have the same state; immutable types must implement equals themselves.

• clone should return an object that has the same state as its object. Immutable types can inherit clone from Object, but mutable types must implement it themselves.

• toString should return a string showing the type and current state of its object. All types must implement toString themselves.
Item 8: Obey the general contract when overriding equals [Bloch]

The equals method implements an equivalence relation. It is:

• Reflexive
• Symmetric
• Transitive
• Consistent
• For any non-null reference value \( x \), \( x.equals(null) \) must return false.
Item 10: Always override `toString` [Bloch]
Queue

• Similar to list
• First In, First Out (FIFO)

• Enqueue
• Dequeue