Plan for Today

- Assignment 2
- Abc test cases
- Data Abstraction
- In-Class Exercise
Signature:
Public static methods (of the Abc class):

\[
\begin{align*}
\text{defg} &: \text{Abc} \times \text{int} \rightarrow \text{int} \\
\text{hijk} &: \text{Abc} \times \text{int} \rightarrow \text{Abc} \\
\text{lmno} &: \text{Abc} \times \text{int} \rightarrow \text{Abc} \\
\text{pqrs} &: \text{int} \rightarrow \text{Abc} \\
\text{tuvw} &: \text{Abc} \rightarrow \text{int}
\end{align*}
\]

Algebraic Specification:

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

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

\begin{align*}
\text{assertTrue("defg f1 1", Abc.defg(f1,1)==3);} \\
\text{assertTrue("defg f1 2", Abc.defg(f1,2)==3);} \\
\text{assertTrue("defg f4 1", Abc.defg(f4,1)==3);} \\
\text{assertTrue("defg f4 2", Abc.defg(f4,2)==4);} \\
\text{assertTrue("defg f4 3", Abc.defg(f4,3)==3);} \\
\text{assertTrue("defg f4 4", Abc.defg(f4,4)==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

\[ \text{data abstraction} = \langle \text{objects}, \text{operations} \rangle \]
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,...,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
        //EFFECTS: If this is empty, throws
        //EmptyException else
        //returns an arbitrary element of
        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