Announcements . . .

- Lectures 1-5 have been updated on the web (fixed a few typos).
- The Java GUI Tutorial will be held in Otto Maass 217 at 18:15.
Last lecture . . .

- A procedure is a mapping from inputs to outputs, with possible modification of inputs.
- Its specification describes its behavior, providing a contract between users and implementors.
- The specification does not change when the implementation changes. This provides locality and modifiability.
- Specifications should have minimal constraining.
- Desirable properties include simplicity and generality.
- Implementations should be total when possible, and may be partial when the context of use is limited and controlled, such as for private helper procedures.
Data Abstraction

- Data abstraction can be achieved through the use of both objects and operations.

- If only objects were provided:
  - The user would implement programs in terms of the data representation.
  - When the representation changed, the program would have to change.

- Therefore, the user has to call the operations to access the data type:
  - When the representation changes, the operation implementations changes.
  - However, the program does not need to change.
Specification of Data Abstractions

- Data types are defined by interfaces and classes:

  ```java
  <visibility> class dname {
    // OVERVIEW: brief description of the date types
    // behavior

    // constructors
    // specs for constructors

    // methods
    // specs for methods
  }
  ```
Components of Specification

- *Overview* gives a description of the abstraction in terms of well understood concepts (e.g. mathematical sets \{ \}, union + ...). It also specifies if the type is mutable or immutable.

- *Constructors* specify how new objects are created.

- *Methods* specify how objects are accessed once they have been created.

- *Constructors* and *Methods* belong to objects, not classes.
  - Because they have no *static* keyword in header.
Abstract Data Type

• To better illustrate the concept of data abstraction, we will use two abstract data type:
  – IntSet
  – Polynomial

• Specification is preliminary version of the class.
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\}
// observers
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
}
• Only one parameterless constructor is enough because the type is mutable.

• Mutators *insert* and *remove* have MODIFIES clause because they modify the object itself (this).

• Observers *isIn*, *size* and *choose* do not change the state of the object.
  
  – Note: Observers are allowed to modify objects other than this, but usually don’t.

• The method *choose* returns an arbitrary element of the IntSet. Thus it is non-deterministic.
public class Poly {
    // OVERVIEW: Polys are immutable polynomials with integer coefficients
    // A typical Poly is c0 + c1x + c2x^2 + c5x^5 + … + cnx^n
    // 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
    }
}
Specification of Poly (cont.)

// methods
public int degree ()
    // EFFECTS: Returns the degree of this, i.e. the largest exponent
    // with a non-zero coef. Returns 0 if this is the zero Poly
public int coeff (int d)
    // EFFECTS: Returns 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 mult (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
}
Comments about Poly

• Poly has two constructors: zero and arbitrary monomial (overloaded)

• Arbitrary polynomials are created by adding and multiplying polynomials, each time creating a new Poly.
  – type is immutable
  – no Mutators
Quiz!

Assuming memory has just been garbage collected and no dead object remains, after the following two statements, how many dead Poly objects does the heap have?

```java
Poly p = new Poly();
p =
p.add((new Poly(5,2)).add((new Poly(3,1)).minus().add(new Poly(9,0))));
```

What will be the representation of the Polynomial in p after the first and after the second statement?
The following function builds an IntSet from a given array.

```java
public static IntSet buildIntSet (int[] a)
    throws NullPointerException {
    // EFFECTS: If p is null throws NullPointerException
    // else returns a set containing an entry for each
    // distinct element of a

    IntSet s = new IntSet();

    for (int i = 0; i < a.length(); i++) {
        s.insert(a[i]);
    }

    return s;
}
```
The following function takes a polynomial and calculates the differential.

```java
public static Poly differential (Poly p) 
  throws NullPointerException {
    // EFFECTS: If p is null throws NullPointerException
    // else returns the Poly obtained by differentiating p

    Poly q = new Poly ();

    for (int i = 1; i <= p.degree(); i++) {
      q = q.add(new Poly(p.coeff(i) * i, i - 1));
    }

    return q;
  }
```
**buildIntSet and differential**

- These functions are not declared in IntSet or Poly, but in another class that uses IntSet and Poly.
- If the implementation of the data abstraction changes, the methods `buildIntSet` and `differential` will continue to work correctly.
- If the methods `buildIntSet` and `differential` are implemented incorrectly, it will not affect the correctness of the abstraction, nor can they break other code that uses the abstraction.
- However, `buildIntSet` and `differential` may be slightly slower that if they were implemented behind the abstraction barrier.
Implementing Data Abstractions

• One data abstraction can have many different possible representations (or reps).

• An implementation makes sure that the representation:
  – is initialized (constructors)
  – used and modified (methods)
  – correctly according to the data abstraction

• A good representation allows all operations to be implemented in a reasonably simple and efficient manner.
  – Frequent operations must run quickly.

• IntSet rep as Vector: allow duplicate elements?
  – insert will be faster
  – remove will be slower
  – isIn will be slower for false, faster for true
Instance variables

- A representation typically has a number of components.
- Each component is stored in an instance variable.
- Instance variables should be declared private:
  - to prevent a user from breaking the abstraction
  - to allow re-implementation without breaking the user’s code
- Instance variables should not be declared static.
  (i.e. there is one of each per object)
- Static variables occur once per class.
  (equivalent to global variables in other languages)
public class Poly {
    // OVERVIEW:
    private int [] trms;
    private int deg;

    // constructors
    public Poly() {
        // EFFECTS: Initializes this to be the zero polynomial
        trms = new int[1];
        deg = 0;
    }
}
public Poly(int c, int n) 
    throws NegativeExponentException {
        // EFFECTS: If n < 0 throws NegativeExponentException else
        // initializes this to be the Poly cx^n
        if (n < 0) 
            throw NegativeExponentException("Poly(int,int) constr");
        if (c == 0) {trms = new int[1]; deg = 0; return;}
        trms = new int[n+1];
        for (int i = 0; i < n; i++) trms[i] = 0;
        trms[n] = c;
        deg = n;
    }

private Poly(int n) {
    trms = new int[n+1];
    deg = 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
    return deg;
}

public int coeff (int d) {
    // EFFECTS: Returns the coefficient of term
    // of this with exponent d
    if (d < 0 || d > deg) return 0;
    else return trms[d];
}
public Poly add (Poly q) 
    throws NullPointerException {
    // EFFECTS: If q is null throws NullPointerException
    // else returns the Poly this + q
    Poly la, sm;
    int i, newdeg;
    if (deg > q.deg) {la = this; sm = q;}
    else {la = q; sm = this;}
    newdeg = la.deg; // new degree is the larger degree
    if (deg == q.deg) // unless there are trailing zeros
        for (int k = deg; k > 0; k--) {
            if (trms[k] + q.trms[k] != 0) break;
            else newdeg--;
        }
    Poly r = new Poly(newdeg); // get a new Poly
    for (i = 0; i < sm.deg && i <= newdeg; i++)
        r.trms[i] = sm.trms[i] + la.trms[i];
    for (int j = i; j <= newdeg; j++)
        r.trms[j] = la.trms[j];
    return r;
}
public Poly minus () {
    // EFFECTS: Returns the Poly - this;
    Poly r = new Poly(deg);
    for (int i = 0; i < deg; i++) r.trms[i] = - trms[i];
    return r;
}

public Poly sub (Poly q)
    throws NullPointerException {
    // EFFECTS: If q is null throws NullPointerException
    // else returns the Poly this - q;
    return add (q.minus());
}
public Poly mul (Poly q) 
   throws NullPointerException { 
   // EFFECTS: If q is null throws NullPointerException 
   // else returns the Poly this * q 

   if ((q.deg == 0 && q.trms[0] == 0) 
       || (deg == 0 && trms[0] == 0)) 
       return new Poly();

   Poly r = new poly(deq + q.deg);
   
   for (int i = 0; i <= deg; i++) 
       for (int j = 0; j <= q.deg; j++)
           r.trms[i+j] = r.trms[i+j] + trms[i] * q.trms[j];
   return r;
}
Poly Implementation

- The Poly representation uses
  - an array storing coefficients (immutable)
  - an integer for storing the degree (for convenience)

- Note that many methods access private instance variables from other objects as well as this.
  (Methods have access to private instance variables of objects of the same class.)

- The method \textit{sub} is implemented in terms of other methods.

- The methods \textit{add}, \textit{mul} and \textit{minus} use private constructor \texttt{Poly(int)} and initialize the new Poly themselves.
Alternative Poly Implementation

• What if most of the terms have zero coefficients?
  – Previous implementation contains mostly zeroes.
  – Maybe we could store only the terms with non-zero coefficients,

• We could solve this problem with 2 vectors:
  – private Vector coeffs; // the non-zero coefficients
  – private Vector exps; // the associated exponents

• However, this is awkward since Vectors have to be precisely lined up.

• Instead, we can use one vector storing both coef and exps.
Records

// inner class
class Pair {
    // OVERVIEW: a record type

    int coeff;
    int exp;

    Pair (int c, int n) {coeff = c; exp = n;}
}

- A record is simply a collection of instance variables and a
  constructor to initialize them. They have no methods.

- You can declare Pair inside Poly as an inner class.

- Do not abuse records. They are only to be used as passive
  storage within a full-blown data abstraction.
Other methods: \emph{equals()} 

- Two objects are equal if they are behaviorally equivalent.
  - it is not possible to distinguish between them using any sequence of calls to the objects

- Mutable objects are equals only if they are the same objects.
  - Otherwise you can change one of them and prove they are not the same
  - equals inherited from Object same as ==

- Immutable objects are equals if they have the same state.
  - They must implement equals themselves.

- Several equals method can be found in an class.
  For example, in the Poly class, we could find :
  - public boolean equals (Poly q)
  - public boolean equals (Object z)
Other methods: \textit{hashCode()}

- The method \textit{int hashCode()} is defined by \texttt{Object}.
- It is used in hash tables to provide a unique number for each distinct object.
- Objects that are equal should have the same \texttt{hashCode}:
  - Mutable objects do not have to define \texttt{hashCode}.
  - Immutable objects have to define \texttt{hashCode}
    (otherwise they will have the same \texttt{hashCode} only if they are \texttt{==})
Other methods: `similar()`

- Two objects are similar if they have the same state at the moment of comparison.
- This is a weaker notion of equality:
  - Similar immutable objects are always equal.
  - Similar mutable objects may not be equal.
- Note that `==` is considered stronger than `equals` and that `equals` is stronger than `similar`.
Other methods: *clone()*

- The method *Object clone()* makes a copy of its object.
- The copy should be similar to the original.
- The default implementation from Object simply makes a new Object and copies all instance variables (shallow copy).
- This is sufficient for mutable objects.
- The method clone() is made accessible by declaring:
  ```java
  public myClass implements Cloneable { . . .
  ```
- Mutable objects should implement their own cloning operation (using a deep copy).
Other methods: `toString()`

- The method `String toString()` should return a String showing the type and current state of the object.
- The default implementation from Object shows type and `hashCode`.
  - This is not very informative.
  - Objects should implement `toString` themselves.
Summary

• Data Abstraction allows us to separate the external interface of an object from its inner working.

• When successful, Data Abstraction allows us to modify the implementation of an object without modifying the other objects using it.

• Differences between mutable and immutable objects.

• Examples with IntSet and Poly.

• Some methods from object may need to be override:
  – equals()
  – similar()
  – hashCode()
  – clone()
  – toString()
Tool of the day: Jikes

- Jikes is a compiler that translate java source files into bytecode.
- In other words, it’s an alternative to javac.
- Why would we need another Java compiler?
  - Open Source: free distribution
  - Strictly Java Compatible: no superset or subset of Java
  - High performance: large projects
  - Dependency analysis: incremental build and makefile generation
- For now, you still need the Sun’s JDK to be installed to have the class libraries.
- Not very user friendly.