Comp-303 : Programming Techniques

Lecture 9

Alexandre Denault
Computer Science
McGill University
Winter 2004

Last lecture . . .

- Adequacy of collection types requires a way to iterate efficiently and conveniently over its elements.
- Iterators provide a that solution.
- A generator object returns elements from the collection one at a time, usually without requiring extra storage or requiring access to all elements.
- Iterators support abstraction by hiding how elements are produced: the generator has access to private variables of the collection but shields the user from this knowledge
- Iterators assume that the collection remains unchanged while iterating, except through the optional remove() operation

Announcements ...

- Deliverables today.
  - Req&Spec Document on my desk.
  - Assignment 1 (paper copy) in the McConnell drop off box (first floor)
  - Assignment 1 (electronic copy) on Web CT
- Assignment 2 will be handed out in 7 days.
- You should start coding the project (2 months left).

February . . .

- Lecture 9: Type hierarchy
- Lecture 10: Polymorphic abstractions
- Lecture 11: Threading
- Lecture 12: Network, sockets and serialization
- Lecture 13: Testing, debugging and review
- Midterm 1

Type hierarchy

- A type family is defined by a type hierarchy.
- At the top of the hierarchy is a supertype that defines behavior common to all family members.
- Other members are subtypes of this supertype.
- A hierarchy can have many levels.
- Type hierarchy can be used
  - to define multiple implementations of a type that are more efficient under particular circumstances.

    DensePoly & SparsePoly implement Poly
  - to extend the behavior of a simple type by providing extra methods

    BufferedReader extends Reader

Substitution principle

- A supertypes behavior must be supported by all subtypes.
- Therefore, in any situation in which a supertype can be used, it can be substituted by a subtype.
- Java compiler enforces this by only allowing extensions to a type (you can only redefine and add methods, not remove them).
- The substitution principle provides abstraction by specification for type hierarchies:
  - Subtypes behave in accordance with the specification in their supertype.

Assignment

- If S is a subtype of T, S objects can be assigned to variables of type T.

  ```
  Poly p1 = new DensePoly(); // the zero Poly
  Poly p2 = new SparsePoly(3,20); // the Poly 3x^20
  ```

- p1 has apparent type Poly
- However, p1, after assignment, has actual type DensePoly.
- The Java compiler checks types based on apparent type.

Method Dispatch in function languages

- At compile time, the destination method is known.
- At compile time, a method call is translated into a jump/branch command to a pre-calculated address.
- We can't do that in Java.

Dispatching

- The compiler cannot determine the actual type of p.

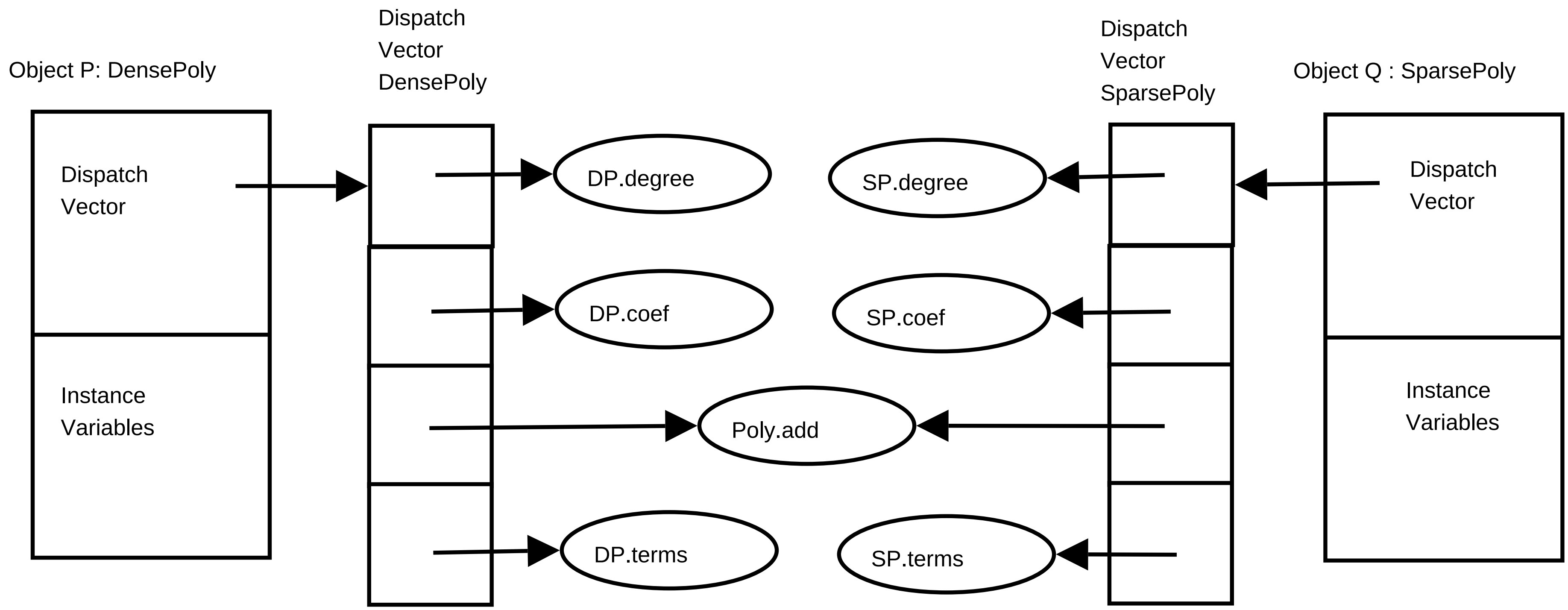
```
static Poly diff (Poly p) {
   // differentiates p
   Iterator g = p.terms();
   ...
}
```

- Therefore the compiler cannot know which method is executed for p.terms()

  (assuming DensePoly and SparsePoly have different implementations for terms())

- The choice of method to call is determined at run time by dispatching the call.

Dispatching (cont.)



- Dispatching maps an method call to the actual type's implementation for that call.
- Each object has a reference to a dispatch vector which stores the implementation for each method.

Method Dispatch in Java

1. Extract the method name and signature from the method call.
2. Search the dispatch vector of the stack object to find our target memory address.
3. Use the method signature to remove arguments from the stack and transfer them to the method's local variables.
4. Transfer control to target memory address.

Method declarations

- Methods can be declared *final*
  - A final method cannot be re-defined (overridden) by a subtype.
  - This guaranties that the behavior of the method is frozen.
- Methods can be declared *abstract*
  - An abstract method must be overridden by a subtype, otherwise the subtype is also an abstract class.
  - Abstract methods are declared but not implemented.
  - This guaranties that subtypes define their own implementation.
- Methods and instance variables can be declared *protected*
  - A protected method is invisible to users (private), but visible to subtypes (public) and classes in the same package.

When to use protected

- Protected methods & instance variables expose the implementation of a supertype to its subtypes.
  - A subtype can break the implementation of a supertype.
  - The supertype can not be re-implemented without affecting the subtype.
  - Protected should only be used when a subtype needs to access parts of its supertype for efficiency reasons.
  - By default, a subtype should access its supertype only through the public interface.

Specification of IntSet

```
public class IntSet {
// OVERVIEW: IntSets are mutable, unbounded sets of integers
// A typical IntSet is {x1,...,xn}

  // 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}
```

Specification of IntSet (cont.)

```
// 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

public void Iterator elements ()
   // EFFECTS: Returns a generator that produces all elements of this
   // (as Integers), each exactly once, in arbitrary order
   // REQUIRES: this not to be modified while the generator is in use
public boolean subset (IntSet s)
   // EFFECTS: Returns true is this is a subset of s else returns false

public String toString ( )
}
```

Partial Implementation of IntSet

```
private Vector els; // the elements

public boolean subset (IntSet s)  {
   // EFFECTS: Returns true is this is a subset of s else returns false

   if (s == null) return false;
   for (int i = 0; i < els.size ( ); i++)
      if (!s.isIn(((Integer) els.get ( i )). intValue ( )))
         return false;
   return true;
}
```

- If *Vector els* was declared protected, we could retrieve integer directly.
- We would not need to call *isIn* multiple times.
- Unfortunately, *subset* would only function with IntSet's that uses *Vector els*.

Subset/Superset Game

I will give you two sets of number. You have to tell me if set B is a subset of set A.

First try

Set A: 4, 19, 10, 500, 38, 32, 203, 401, 134, 3, 54, 27, 76, 348, 122, 453, 88, 95, 499, 176, 365, 9, 473, 112, 62, 201, 465, 333, 67, 262

Set B: 465, 401, 348, 88, 122, 95, 176, 262, 10, 134, 27, 9, 67

Second try

Set A: 1, 3, 4, 9, 10, 19, 26, 32, 38, 53, 62, 67, 76, 88, 99, 112, 142, 154, 186, 201, 213, 262, 333, 358, 365, 401, 453, 455, 476, 499, 500

Set B: 9, 10, 67, 99, 201, 333, 365, 401, 453, 477, 499, 500

SortedIntSet

- Suppose we define a type SortedIntSet, which is like an IntSet but gives access to its elements in sorted order by generator returned by terms().
- The method *subset()* could have a more efficient implementation, since we can assume elements are stored in sorted order.

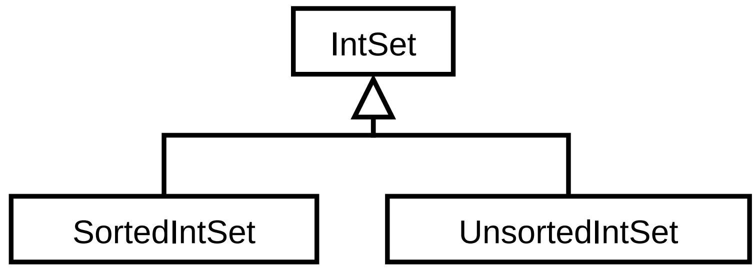
  We overload subset:

```
public boolean subset (IntSet s) // inherited
public boolean subset (SortedIntSet s) // overloaded
   // same specification but more efficient:
   // go through elements of this and s
   // from small to large
```

Implementing SortedIntSet

- To implement SortedIntSet we could use a sorted list. This could make SortedIntSet much more efficient.
- If SortedIntSet is a subclass of IntSet, every SortedIntSet object will have all instance variables declared for IntSet.
- However, we do not need a *Vector els* variable in our SortedIntSet.
- This means we might need to rethink the structure of our classes.

Abstract IntSet



- IntSet is now an abstract class.
- Methods common to all IntSets are store in IntSet.
- Implementation details specific to particular type of IntSet are store in the subtypes.

Abstract IntSet (cont.)

```
public abstract class IntSet {
   protected int sz; // the size

   public IntSet () { sz = 0; }

   public abstract void insert (int x);
   public abstract void remove (int x);
   public abstract Iterator elements ( );

   public boolean isIn (int x) {
      Iterator g = elements ( );
      Integer z = new Integer (x);
      while (g.hasNext ( ))
         if (g.next ( ). equals (z )) return true;
      return false;
   }

   public int size () { return sz; }
}
```

Abstract IntSet (cont.)

- Instance variable *sz* is provided to enable all subclasses to implement size ( ) efficiently.
- The variable *sz* is declared protected so subclasses can access (read/write) it.
- The methods *insert()*, *remove()* and *elements()* are abstract.
- The methods *isIn()*, *subset()* and *toString()* are implemented using abstract methods.
  - These methods are called *template* since they implement behavior in an abstract class using calls to abstract methods.
- The object IntSet() can not be called by user since this is an abstract class (no instances).
- However, subclasses can call the constructor using the *super()* keyword.

Interfaces

- An *abstract* class defines a type and provides a partial implementation such as instance variables and template methods.
- An *interface* defines only a type. In other words, it is composed of public nonstatic abstract methods.
- We can use interfaces with the keyword implements in the header of the class.

Iterator interface

```
public interface Iterator {

   public boolean hasNext ( );
      // EFFECTS: Returns true if there are no more
      // elements to yield else returns false

   public Object next ( ) throws NoSuchElementException;
      // MODIFIES: this
      // EFFECTS: If there are more results to yield, returns
      // the next result and modifies the state of this to
      // to record the yield.
      // Otherwise, throws NoSuchElementException
}
```

Multiple Interfaces

- Interfaces are used when all methods are abstract.
- Interfaces can be used when a type has multiple supertypes.
- For example, an object can implement an *iterator* and a *cloneable* interface.
- You can mix both inheritance and interfaces.
- For example, an object can implement a *cloneable* interface and extend *IntSet*.
- You can only extend one class, but implement multiple interfaces.

Multiple Implementations

- Type hierarchy can be used to provide multiple implementations of a type.
- Subclasses only override methods defined in the (abstract) superclass.
- They do not add public methods (private helper methods are allowed).
- Constructors must be defined and could be overloaded with extra parameters.
- User code is defined only in terms of the supertype, except when creating objects.
  - The user is unaware whether a Poly is a DensePoly or a SparsePoly
- The actual type can change behind the users back.

Meaning of subtypes

- The substitution principle requires that the subtype specification supports reasoning based on the supertype spec. Three properties must hold:
  - *Signature rule*: the subtype must have all methods of the supertype, and the signatures of subtype methods must be compatible with the signatures of supertype methods.
  - *Methods rule*: calls of subtype methods must behave like calls to corresponding supertype methods.
  - *Properties rule*: the subtype must preserve all properties that can be proven about supertype objects.

Signature rule

- The signature rule is checked by the Java compiler:
  - The subtype must have all supertype methods with identical signatures.
  - However, a subtype method can have fewer exceptions.
- Code written in terms of supertype can handle exceptions listed in supertype methods but will work in a type-correct manner if those exceptions are not thrown.
- Java's compatibility is stricter than necessary.

Methods rule

- Behavior can not be checked by the compiler. The programmer must guarantee that subtype methods behave like supertype methods.
  - With any IntSet object, the method *insert(x)* should add a number to the set.
- In all our examples, subtype methods behave like supertype methods, except for the *elements()* method of SortedIntSet.
  - The method *elements()* is underdetermined in our supertype. Requiring *elements()* to return a generator that gives elements in sorted order narrows the specification but is still correct.

Allowable changes in specs

- A subtype method can have weaker preconditions and stronger postconditions.
  - Precondition rule: the set of possible input for the subtype if bigger or equal than the set of possible input for the supertype
  - Postcondition rule: the set of possible output for the subtype if smaller or equal than the set of possible output for the supertype

Allowable changes in specs (cont.)

- Weakening precondition allows subtype to require less from its caller than the supertype.
  - It allows code written in terms of supertype to execute correctly.
- Strengthening postcondition allows subtype to restrict the result more than supertype
  - It allows code written in terms of the supertype to execute correctly, since postcondition of supertype follows from postcondition of the subtype.

The properties rule

- Properties defined for supertype must hold for subtype.
- The invariant properties always hold: if supertype is immutable, subtype must be immutable.
- Some related objects cannot be subtyped because of their properties.
  - For example, a SimpleSet is a IntSet which can only grow.
  - IntSet cannot be a subtype of SimpleSet since it has a remove() method and can therefore shrink.
  - SimpleSet cannot be a subtype of IntSet since the remove() method can only be inherited or overridden with preservation of the methods rule.

Summary

Type hierarchies improve the structure of programs.

- By *grouping* types into a family, the programmer makes clear that there is a relationship between them. This makes code easier to understand than when all types are defined in a flat structure.
- Hierarchy allows *definition* of abstractions that work for an entire family. This allows user code that works for all Windows, for example , regardless of the specific type of Window that is passed as argument.
- Hierarchy provides *extensibility*. New types can be defined and will be handled correctly by user code that was written even before the new types were invented.