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.
Polymorphic Abstractions

- Collections like IntSet are defined for one type only (IntSet and its subtypes).
- It is more convenient to defined collections that work for many different object types.
- Polymorphism generalizes abstractions so that they work for many types.
  - They allow us to avoid redefining abstractions.
  - Instead a single abstraction becomes more widely useful (similar to abstraction by parameterization).
- A procedure or iterator can be polymorphic with respect to the types of one or more arguments.
- A data abstraction can be polymorphic with respect to the types of the elements its objects contain.
public class Set {
    // OVERVIEW: Sets are mutable, unbounded sets of objects
    // null is never an element of a Set. The Equals method is used to
    // determine equality of elements

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

    // methods
    public void insert (Object x) throws NullPointerException
        // MODIFIES: this
        // EFFECTS: If x is null throws NullPointerExceptions else
        // adds x to the elements of this, i.e. this_post = this + {x}
public void remove (Object x)
    // MODIFIES: this
    // EFFECTS: If x is in this, removes x from this else
    // does nothing
public boolean isIn (Object x)
    // EFFECTS: if x is in this returns true else returns false
public boolean subset (Set s)
    // EFFECTS: If all elements of this are elements of s
    // return true
    // else returns false

    // Specifications of size() and elements()
Set

- Specification is similar to IntSet, but all methods take Object as argument instead of int.

- The specification mentions which method is used to determine equality: *Object.Equals()*.
Partial implementation of Set

private Vector els; // the elements

public Set () { els = new Vector (); }
private Set (Vector x) { els = x; }

public Object clone () {
    return new Set ((Vector) els.clone ());
}
Partial implementation of Set

```java
public void insert (Object x) throws NullPointerException {
    if (getIndex ( x ) < 0) els.add( x ) ;
}

private int getIndex ( Object x) {
    for (int i = 0; i < els.size( ); i++)
        if (x.equals(els.get( i )) return i ;
    return -1;
}
```
Set implementation

• The method `insert()` stores its argument object in the set, not a clone, as indicated in specification.

• If a collections stores clones, the specification must mention this.

• The clone method clones the set but not the contained objects.

• This does not expose the representation since the state of the set is determined by the identity of the contained objects, not their state.
Using Polymorphic Abstractions

- Similar to non-polymorphic abstractions, but only \textit{Objects} can be stored, so primitive values like \textit{int} must be wrapped:
  
  \begin{verbatim}
  s.insert( new Integer(3) );
  \end{verbatim}

- Elements are returned as Objects so the using code must cast to the expected type and, if necessary, unwrap a primitive value.
  
  \begin{verbatim}
  while (g.hasNext()) {
    int i = ((Integer) g.next()).intValue();
  }
  \end{verbatim}

- The compiler cannot guarantee that the objects stored in a polymorphic collection are of the expected type.

- This transforms compile-time type errors into run-time \texttt{ClassCastException}.

February 24, 2004  Lecture 10 – Comp 303 : Programming Techniques
Equality

- Two Vectors that have the same state are considered equal by the default Equals() method (defined for many collections).
- This can be a problem when storing Vectors in a set.

```java
Set s = new Set();

Vector x = new Vector();
Vector y = new Vector();
s.insert(x);
s.insert(y);       // y is not added: it appears to be in s
x.add(new Integer(3));
if (s.isIn(y))      // returns false
```
Fix by wrapping vectors in container objects

- Now we must always wrap objects in containers and pass those as arguments to the Set.

```java
Set s = new Set();
Vector x = new Vector();
Vector y = new Vector();

s.insert(new Container(x));
s.insert(new Container(y));

x.add(new Integer(3));
if (s.isIn(new Container(y))) // returns true because
    // Container.equals() uses
    // == to determine equality
```
Other methods used in Polymorphic Collections

• Set only relies on the `Object` interface because it only requires the `Equals()` method.

• Other polymorphic collections might require other methods.
  – Sorted or ordered collections require a way to compare two elements.
  – The Interface `Comparable` specifies such elements.
  – This is called the *element subtype approach*.
  – Objects than implement `Comparable` can be sorted when used with Java collections.
public interface Comparable {
    // OVERVIEW: Subtypes of Comparable provide a method
    // to determine the ordering of their objects. This ordering
    // must be a total order over their objects, and it should be
    // both transitive and symmetric.
    // x.compareTo( x ) == 0 => x.equals( y)

    public int compareTo (Object x)
        throws ClassCastException, NullPointerException;
    // EFFECTS: If x is null, throws NullPointerException
    // if this and x are not comparable,
    // throws ClassCastException
    // Otherwise, if this is less than x returns - 1
    // if this equals x returns 0
    // if this is greater than x returns 1
public class OrderedList {
    // OVERVIEW: An OrderedList is a mutable ordered list of Comparable
    // Objects. A typical list is the sequence [x1,,xn] where xi < xj if i < j

    private boolean empty;
    private OrderedList left, right;
    private Comparable val;

    // constructor
    public OrderedList () {
        // EFFECTS: Initializes this to be an empty ordered list.

        empty = true;
    }
}
Partial Specification of OrderedList

// methods
public void addEl ( Comparable el )
    throws NullPointerException, DuplicateException, ClassCastException;
    // MODIFIES: this
    // EFFECTS: If el is in this, throws DuplicateException
    // if el is null throws NullPointerException
    // if el cannot be compared throws ClassCastException
    // otherwise adds el to this

public boolean isIn (Comparable el)
    // EFFECTS: if el is in this returns true else returns false
public void addEl ( Comparable el )
throws NullPointerException, DuplicateException, ClassCastException{
    if (val = null) throw new NullPointerException("OrderedList.addEl");
    if (empty) {
        left = new OrderedList( );
        right = new OrderedList( );
        val = el;
        empty = false;
        return;
    }
    int n = el.compareTo( val );
    if (n == 0) throw new DuplicateException( "OrderedList.addEl" );
    if (n < 0) left.addEl (el);
    else right.addEl (el);
}
• OrderedList requires that types are homogeneous: once an element is added, any new added element must be comparable to it, otherwise a ClassCastException is thrown.

• When the OrderedList becomes empty, a new, different element type can be added, and from then on, all added elements must be comparable to that new element.
More flexibility

- It is sometimes convenient to define a different ordering for existing elements, for instance, to create an OrderedList in which elements are sorted in reverse order.
- The order is an attribute of the collection, not of its elements.
- Java.util provides a Comparator interface for this purpose.
- This is called the related subtype approach.
public interface Comparator {
    public int compare (Object x, Object y)
        throws ClassCastException, NullPointerException;
    // EFFECTS: If x or y is null, throws NullPointerException
    // if x and y are not comparable, throws ClassCastException
    // Otherwise, if x is less than y returns - 1
    // if x equals y returns 0
    // if x is greater than y returns 1
}
public class OrderedList {
    // OVERVIEW: An OrderedList is a mutable ordered list of Comparable
    // Objects. A typical list is the sequence [x1,,xn] where xi < xj if i < j

    // constructors
    public OrderedList () ;
        // EFFECTS: Initializes this to be an empty ordered list.
        // The order is defined by the "natural" order defined on elements
        // that implement the Comparable interface.
    public OrderedList ( Comparator c) ;
        // EFFECTS: Initializes this to be an empty ordered list.
        // The order is defined by Comparator c which supercedes
        // the elements natural ordering.

    // methods
    public Comparator comparator () ;
        // EFFECTS: Returns the comparator associated with this sorted set,
        // or null if it uses its elements’ natural ordering
}
More flexibility

- The Comparator trick can be used to extend the functionality of a collection after the element types have been defined (even if the element types can not be subtyped, for instance if they have been declared final).

- The collection will use the added interface for the extra operations required by the collection.

  For each possible element type, a new implementation for the interface must be provided.

- For example, a collection SumSet which keeps track of the sum of its elements.
public class SumSet {
    // OVERVIEW: SumSets are mutable sets of objects plus a sum
    // of all current objects in the set. The sum is computed using an Adder
    // object. All elements of the set are addable using the Adder

    private Vector els; // the elements
    private Object s; // the sum of the elements
    private Adder a; // the object used to do adding and subtracting

    // constructor
    public SumSet (Adder p) throws NullPointerException {
        // EFFECTS: Makes this to be empty set whose elements can be
        // added using p, with initial sum p.zero

        els = new Vector ( );
        a = p;
        s = p.zero ( );
    }
}
public void insert ( Object x )
    throws NullPointerException, ClassCastException{
    // MODIFIES: this
    // EFFECTS: If x is nul throws NullPointerException
    // if x cannot be added throws ClassCastException
    // else adds x to the set and adjusts the sum

    Object z = a.add( s, x );
    int i = getIndex( x );
    if (i < 0 ) {
        els. add( x );
        s = z;
    }
}
public interface Adder {
    // OVERVIEW: All subtypes of Adder provide a means to add
    // and subtract the elements of some related object type

    public Object add ( Object x, Object y )
        throws NullPointerException, ClassCastException ;
    // EFFECTS: If x or y is null throws NullPointerException
    // if x and y are not addable throws ClassCastException
    // else returns the sum of x and y

    public Object sub ( Object x, Object y )
        throws NullPointerException, ClassCastException ;
    // EFFECTS: If x or y is null throws NullPointerException
    // if x and y are not addable throws ClassCastException
    // else returns the difference of x and y

    public Object zero( );
    // EFFECTS: returns the object that represents zero for the
    // related type.
}

public class PolyAdder implements Adder {

    private Poly z; // the zero Poly

    public PolyAdder ( ) { z = new Poly ( ); }

    public Object add ( Object x, Object y ) throws NullPointerException, ClassCastException {
        if (x == null || y == null) throw new NullPointerException ("PolyAdder.add");
        return ((Poly) x).add((Poly) y);
    }

    public Object sub ( Object x, Object y ) throws NullPointerException, ClassCastException {
        if (x == null || y == null) throw new NullPointerException ("PolyAdder.sub");
        return ((Poly) x).sub((Poly) y);
    }

    public Object zero ( ) { return z; }
}

Using SumSet

Adder a = new PolyAdder ( );
SumSet s = new SumSet ( a );
s.insert (new Poly (3, 7));
s.insert (new Poly (4, 8));
Poly p = (Poly) s.sum ( );
Polymorphic procedures

• Polymorphic abstraction techniques can also be used for procedures.

• The techniques are similar:
  – Requires elements to implement an interface with the required methods such as Addable or Comparable.
  – Provides an object independent from the manipulated types. The object implements an interface that specifies the required methods.
Polymorphic procedures on Vectors

```java
public static sort (List list) throws ClassCastException
    // MODIFIES: list
    // EFFECTS: if list is not null, sorts it into ascending order using
    // the compareTo method of Comparable. If some element of list is
    // null or not comparable throws ClassCastException.

public static sort (List list, Comparator c) throws ClassCastException
    // MODIFIES: list
    // EFFECTS: if list is not null, sorts it into ascending order using
    // the compare method of c. If some element of list is null or not
    // comparable using c throws ClassCastException.
```
Summary

• Polymorphic abstractions provide a way to abstract from the types of parameters.

• Procedures, Iterators and data abstractions can benefit from polymorphic abstraction.

• A polymorphic abstraction usually requires certain methods to apply to the parameters.

• In the element subtype approach, an interface defines these methods which take a parameter object as receiver (define a supertype).

• In the related subtype approach, an interface defines a new object type which has these methods that take parameter objects as arguments.
Tool of the day: Air Conditioners

- Today’s tool of the day is an IBM home air conditioner.
- This is not a joke.
- IBM and air-conditioning maker Carrier were introducing the remote-controlled air conditioner.
- Owners were able to turn on the chiller by logging on to the Myappliance.com Web site.
- The Carrier air conditioner carries an embedded Java chip, and runs IBM software and hardware.
- The technology was tested in Europe back in 2001.