Lecture Notes
Chapter #9
Summery

Inheritance

Subclass

- **inherits all members of its superclass except its constructors**
- **methods inherited from a superclass can overridden in the subclass**
    - **a method overrides another method if both methods have the same declaration**
- **can access private members of superclass only by use of the prefix "super. <member_name>", which is a reference to the superclass, i.e., super.displayStatistics( ) refers to the method displayStatistics( ) which resides in the superclass while displayStatistics( ) refers to the overridden method in the subclass**

Annotations

- **notification to the compiler of special circumstances which may require special handling**
- **@Override is an annotation to the compiler to obviate the accidental creation of a new method name by programmer error, i.e., misspelling or parameter list variations**

Access Modifiers **figure 9-4**

- **Public** **members are visible to**
  clients, subclasses, package, namespace
- **Private** **members are visible to**
  itself only
- **Protected** **members are visible to**
  subclasses and classes in the same package
- **Package** **(default or no modifier is specified)**
  members are visible to
  subclasses, classes in the same package

Relationships

- **Is-A**
    - **Inheritance**
    - **Superclass-Subclass**
    - **Object Type Compatibility (Upward Only)**
        - **a subclass is type compatible with its superclass**
        - **a superclass is NOT type compatible with its subclass**
        - **parameter placeholder "Object"**
- **Has-A**
    - **Containment**
        - **StackListBased (Ch 7) contains ListReferenceBased**
        - **QueueListBased (Ch 8) contains ListReferenceBased**

Polymorphism

Dynamic Binding, i.e., Late Binding

- **Ball is a subclass of Sphere**
- **Sphere defines method displayStatistics( )**
- **Ball overrides inherited method displayStatistics( )**
- **mySphere is an instance of Sphere**
- **myBall is an instance of Ball**

- **given a method y(Object)**
  - **y(mySphere) will invoke mySphere.displayStatistics( )**
  - **y(myBall) will invoke myBall.displayStatistics( )**

- **given**
  - **Ball myBall = new Ball(1.25, “golfball”);**
  - **Sphere mySphere = myBall;**
  - **mySphere.displayStatistics( );**

  mySphere actually references an instance of Ball,
  hence myBall.displayStatistics( ) IS EXECUTED

- **y(Object) is a POLYMORPHIC METHOD**
  - **the type of <u>method to execute</u> or <u>object to create</u> is**
    DETERMINED AT RUN-TIME

Compile-Time Binding, Early Binding, Static Binding

- **the type of <u>method to execute</u> or <u>object to create</u> is**
  DETERMINED AT COMPILE-TIME
- **field modifier <u>final</u> attached to a superclass method definition requires the compiler to BIND THE METHOD AT COMPILE-TIME hence the method cannot be overridden by a subclass**

- **field modifier <u>static</u> attached to a method definition requires the compiler to BIND THE METHOD AT COMPILE-TIME**

- **field modifier <u>abstract</u> requires the subclass to override the method**

Overriding Methods

- **create method with the same name and same parameter list as the original method**

Overloading Methods

- **create method with the same name but with a different parameter list as the original method**

Abstract Classes

- **has no instances**
- **used only as the basis of other classes**
- **can contain both data fields and methods**
- **declaration**
    - **insert keyword *abstract* in definition**
    - **include abstract method, i.e., a method without a body, in the class**
- **any subclass that fails to implement an abstract method is, itself, an abstract class**
- **protected data fields within an abstract class is reasonable since it is always a superclass of a subclass**
- **constructors cannot be abstract**
- **should omit implementations of all methods except those that provide access to private data fields or that express functionality common to all subclasses**

Interfaces

- **specify common behaviors to a set of classes which need not be necessarily related**
- **specify the methods for a class, e.g.,**
    - **StackInterface stack = new StackArrayBased( )**
    - **StackInterface stack = new StackReferencedBased( )**
    - **use only StackInterface methods, e.g., stack.pop( );**

  enables movement from one implementation to the other by simply changing the creation statements

Subinterfaces

JCF

- **java.util.Iterable**
- **java.util.Collection**

Generics

Generic Classes

```
public class NewClass <T>
{
  private int year;
  private T data = null;
          ⋮

}


static public void main( String [ ] args )
{
  NewClass<String> first = new NewClass<String> ("Wally", 2010);
  NewClass<Integer> second = new NewClass<Integer>( 15 );
          ⋮

}
```

- primitive types are not allowed as generic type parameters
- generic types cannot be used in array declarations, use JCF
  - Vector<T> test = new Vector<T>( );
  - ArrayList<T> test2 = new ArrayList<T>( );

Generic Wildcards

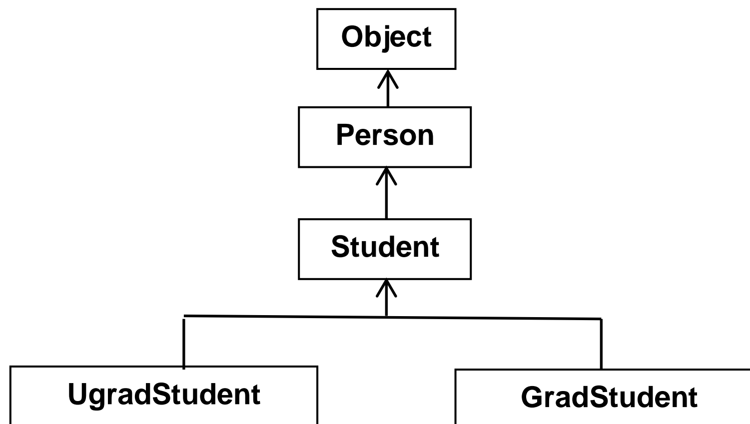
- " ? " wildcard is used to create a method which can accept both the <u>first</u> and <u>second</u> objects

```
public void process( NewClass<?> temp )
{
  System.out.println( "getData( ) => " + temp.getData( ) );
}
```

Generic Class Inheritance

```
public class Book<T, S, R>
public class RuleBook<T, S, R> extends Book<T, S, R>
public class myBook extends Book<Integer, String, String>
public class TextBook<T, Q> extends Book<T, String, String>
```

- subclass method overrides a superclass method if
  - they have the same parameters
  - return type of the subclass method is a <u>subtype of all the methods</u> that it overrides



Binding Generic Data Types

```
public interface Registration <T extends Student>
{
  public void register( T student, CourseID cid);
  public void drop( T student, CourseID cid);
                    ⋮
}
```

VALID

- Registration<Student> students = new Registration<Student>( );
- Registration<UgradStudent> ugrads = new Registration<UgradStudent>( );
- Registration<GradStudent> grads = new Registration<GradStudent>( );

INVALID generates an error message

- Registration<Person> people = new Registration<Person>( );

Binding the Wildcard "?"

```
public void process ( ArrayList< ? extends Student> stuList )
```

VALID

```
ArrayList<UgradStudent> ugList = new ArrayList<UgradStudent>( );
test1.process(ugList);
```

INVALID generates an error message

```
ArrayList<Person> pList = new ArrayList<Person>( );
test1.process(pList);
```

Specification of a Type that is too Restrictive

```
class Student extends Person implements Comparable<Student>
{
  protected String id;
                        ⋮
  public int compareTo( Student s )
  {
    return id.compreTo(s.id);
  }
                        ⋮
}
```

Assume

```
import java.util.ArrayList;
class MyList <T implements Comparable<T>>
{
  ArrayList<T> list = new ArrayList<T>( );

  public void add( T  x )
  {
                        ⋮
     if ((List.get( I )).compareTo(list.get( j )) < 0 )
     {
                        ⋮
     }
  }
                        ⋮
}
```

MyList<Student> it320 = new MyList<Student>( ); **COMPILES**

MyList<UgradStudent> it321 = new MyList<UgradStudent>( ); **DOES NOT COMPILE**

Redefine
class MyList <T implements Comparable< ? super T>>

< ? super T>

- **specifies a lower bound on the data-type parameter**
- **allows the class or a superclass to be used as the actual data-type parameter**

Generic List Class

Reference Based List Class

package List: pages 480 – 481 cf pages 265 -- 268

```
public interface ListInterface<T>
{
  public boolean isEmpty( );
  public int size( );
  public void add( int index, <T> ) throws ListIndexOutOfBoundsException;
  public void remove( int index ) throws ListIndexOutOfBoundsException;
  public <T>  get( int index ) throws ListIndexOutOfBoundsException;
  public removeAll( );
}
package List:
public class ListReferenceBased<T>  implements ListInterface<T>
{
  private Node<T>  head;
  private int numItems;
            ⋮
}
```

private data items
private methods
public methods

- Define data types ➔ -- filled: Boolean
- Specify abstract methods ➔
  - double area (double length, double width) = 0;
  - public Boolean isFilled();

Generic Methods

```
public static <T extends Comparable<? super T>>
void sort( ArrayList<T> list )
{
            ⋮
}
```

The compiler determines the method data-types by using the actual data-type arguments provided in the method invocation

```
class TestMethod
{
  public static void main( String[ ] args )
  {
    ArrayList<String> names = new ArrayList<String> ( );
    names.add("Janet");
    names.add("Andrew");
    names.add("Sarah");
            ⋮
  }
}
```

Array Based ADT List Class

+createList( ) pages 226 -- 233
+isEmpty( ) : boolean
+size( ) : integer
+add(in index : integer, in newItem : ListItemType)
+remove(in index : integer)
+removeAll( )
+get(in index : integer) : ListItemType

```
public interface BasicADTInterface            pages 484 – 489
{
  public int size( );
  public boolean isEmpty( );
  public void removal( );
}
```

```
public interface ListInterface<T> extends BasicADTInterface
{
  public void add( int index, T item ) throws ListIndexOutOfBoundsException;
  public void remove( int index ) throws ListIndexOutOfBoundsException;
  public T  get( int index ) throws ListIndexOutOfBoundsException;
}
```

Array Based ADT Sorted List Class

+createSortedList( )
+isEmpty( ) : Boolean {query}
+size( ) : integer {query}
+sortedAdd(in newItem : ListItemType) throw ListException
+sortedRemove(in newItem : ListItemType) throw ListException
+removeAll( )
+get(in index : integer) : ListItemType throws ListIndexOutOfBoundsException;

```
public interface
    SortedListInterface<T extends Comparble<? Super T>> extends BasicADTInterface
    {
      public void sortedAdd( T  newItem ) throws ListException;
      public T  get( int index ) throws ListIndexOutOfBoundsException;
      public int locateIndex( T anItem );
      public void sortedRemove( T anItem ) throws ListException;

    }
```

IS-A Sorted List is a List

```
public class SortedList <T extends Comparable<? super T>>
            extends ListReferenceBased<T>
            extends SortedlistInterface<t>
{
    . . .
  public void sortedAdd( T newItem )
  {
    int newPosition = locateIndex( newItem );
    super.add( newPosition, newItem );
  }

  public void sortedRemove( T anItem ) throws ListException
  {
    int newPosition = locateIndex( newItem );
    if (( anItem.compareTo( get( position )) == 0))
    {
      super.remove(position);
    }
    else
    {
      throw new ListException( "Sorted Remove Failed" );
    }
  }
            . . .
}
```

HAS-A Sorted List has a List

Use an instance of an existing class to implement a new class

```
public class SortedList <T extends Comparable<? super T>>
            implements SortedListInterface<T>
{
  private ListInterface<T> aList;

  public sortedList( )
  {
    aList = new ListReferenceBased<T>) );
  }
            . . .
  public void sortedAdd( T newItem )
  {
    int newPosition = locateIndex( newItem );
    aList.add( newPosition, newItem );
  }

  public void sortedRemove( T anItem ) throws ListException
  {
                    . . .
  }
            . . .
}
```

Iterators

```
public interface ListIterator<E> extends Iterator<E>
{
  void add( E o );
  boolean hasNext( );
  boolean hasPrevious( );
  E next( ) throws NoSuchElementException;
  Int nextIndex( );
  E previous( ) throws NoSuchElementException;
  int previousIndex( );
  void remove( ) throws UnsupportedOperationException, IllegalStateException;
  void set( E o ) throws UnsupportedOperationException, IllegalStateException;
}
```

Implementation

```
public class MyListIterator<T> implements java.util.ListIterator<T>
{
  private ListInterface<T> list;
  private int cursor;
  private int lastItemIndex;

  public MyListIterator( ListInterface<T> list)
  {
    this.list = list;
    cursor = 0;
    lastItemIndex = -1;
  }

  public void add( T item )
  {
    list.add(cursor + 1, item );
    cursor++;
    lastItemIndex = -1;
  }
       . . .
}
```

see pages 492 -- 493