COT 3530: Data Structures

Giri Narasimhan
ECS 389; Phone: x3748
giri@cs.fiu.edu
www.cs.fiu.edu/~giri/teach/3530Spring04.html

Figure 2.15, page 53

```
import java.io.InputStreamReader;
import java.io.BufferedReader;
import java.io.IOException;
import java.util.StringTokenizer;
public class MaxTest
{
    public static void main( String [ ] args )
    {
        BufferedReader in = new BufferedReader( new InputStreamReader( System.in ) );
        String oneLine;
        StringTokenizer str;
        int x, y;

        System.out.println( "Enter 2 ints on one line: " );
        try
        {
            oneLine = in.readLine( );
            if( oneLine == null )
                return;

            str = new StringTokenizer( oneLine );
            if( str.countTokens( ) != 2 )
            {
                System.out.println( "Error: need two ints" );
                return;
            }
            x = Integer.parseInt( str.nextToken( ) );
            y = Integer.parseInt( str.nextToken( ) );
            System.out.println( "Max: " + Math.max( x, y ) );
        }
        catch( IOException e )
          { System.err.println( "Unexpected IO error" ); }
        catch( NumberFormatException e )
          { System.err.println( "Error: need two ints" ); }
    }
}
```

Figure 2.16, page 54

Command Line Arguments

Reading from Sequential Files

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.IOException;

public class ListFileContents
{
   public static void main( String [ ] args )
   {
      if( args.length == 0 ) System.out.println( "No files specified" );
      for( int i = 0; i < args.length; i++ )  listFile( args[ i ] );
   }
   public static void listFile( String fileName )
   {
      FileReader theFile;
      BufferedReader fileIn = null;
      String oneLine;
      System.out.println( "FILE: " + fileName );
      try
      {
         theFile = new FileReader( fileName );
         fileIn  = new BufferedReader( theFile );
         while( ( oneLine = fileIn.readLine( ) ) != null )
            System.out.println( oneLine );
      }
      catch( IOException e )
       {  System.out.println( e ); }
      finally
      {
         // Close the stream
         try
         {
            if (fileIn != null )  fileIn.close( );
         }
         catch( IOException e ) {  }
      }
   }
}
```

Figure 2.17, page 56

Writing to Sequential Files

```
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;
public class DoubleSpace
{
   public static void main( String [ ] args )
   {
      for( int i = 0; i < args.length; i++ )
         doubleSpace( args[ i ] );
   }
   public static void doubleSpace( String fileName )
   {
      PrintWriter   fileOut = null;
      BufferedReader fileIn = null;
      try
      {
         fileIn  = new BufferedReader( new FileReader( fileName ) );
         fileOut = new PrintWriter(  new FileWriter( fileName + ".ds" ) );
         String oneLine;
         while( ( oneLine = fileIn.readLine( ) ) != null )
            fileOut.println( oneLine + "\n" );
      }
      catch( IOException e )      { e.printStackTrace( ); }
      finally
      {
         try
         {
            if( fileOut != null )     fileOut.close( );
            if( fileIn != null )       fileIn.close( );
         }
         catch( IOException e )
          { e.printStackTrace( ); }
      }
   }
}
```

Objects & Classes

- Difference between a class and an object.
  - An object is an instance of a class.
- Private, public, protected, package visibility
- Basic methods:

  constructors, mutators, assessors, output, equals.
- Expression to check type of object: instanceof.
- Reference to current object & constructor: this.
- Global constant: static final.

Packages

- Group of related classes.
- Specified by package statement.
- Fewer restrictions on access among each other;
  - if class is called public, then it is visible to all classes
  - if no visibility modifier is specified, its visibility is termed as "package visibility" and is somewhere between:
    - private (other classes in package cannot access it) and
    - public (other classes outside package can also access it)
- Package locations can be specified by environmental variables.

Defining a Class

- The Student class describes a single student. It contains a single instance field named lastName
- Each Student object will have a unique copy of its own instance fields

```
public class Student {
  String lastName;
}
```

Declaring an Object

- The Student class describes a single student. It contains a single instance field named lastName
- Each Student object contains a distinct copy of its instance fields

```
Student first = new Student();

Student secnd = new Student();
```

first

secnd

Add a Constructor

- Executed when an object is created
- Same name as the class
- No return type
- Without parameters, it is called a default constructor

```
public class Student {
  public Student()
  {
    lastName = "(none)";
  }

  String lastName;
}
```

Add a toString Method

- The toString() method is already defined in the Object class
- We can provide our own version here

```
class Student {

   Student()
   {
     lastName = "Smith";
   }

   public String toString()
   {
     return "Last name = " + lastName;
   }

   String lastName;
}
```

Add a Public Test Class

- Every program must have a public class that contains main()
- Keep this class short and simple
- Static method does not need "controlling" object. Static field is shared by all instances of that class.

```
public class StudentTest {

  public static void main( String args[] )
  {
    Student S = new Student();
    System.out.println( S.toString() );
  }
}

// (See the Student1 project)
```

Add a Second Constructor

- This constructor has a String parameter that initializes the lastName instance field

```
public class Student {

  public Student( String aName )
  {
    lastName = aName;
  }

}
```

Selectors and Mutators

- A selector method returns the value of an instance field
- A mutator method changes the value of an instance field

```
public String getLastName()
{
  return lastName;
}

public void setLastName( String aName )
{
  lastName = aName;
}
```

Selectors and Mutators

- A selector method returns the value of an instance field
- A mutator method changes the value of an instance field

```
Student S2 = new Student("Ramakrishnan");

S2.setLastName("Chong");

System.out.println( "New name of S2: "
       + S2.getLastName() );
```

Using the JavaDoc Utility

- JavaDoc generates HTML documentation for your public classes and methods
- Use the /** delimiter to begin a comment, and */ to end
- Appears before classes and methods

```
/**
   A class that holds information about a single
   college student. Demonstrates an overloaded
   constructor.
*/

public class Student {
. . .
```

Using the JavaDoc Utility - 2

- Run JavaDoc from the Tools menu in JCreator
- To install JavaDoc: follow instructions on my Samples page.

```
/**
   Program entry point; creates two students
   with different names.
*/

public static void main( String args[] )
{
   . . .
```

Using the JavaDoc Utility - 3

- @param – Identifies a method parameter
- @return – describes the function return value.

```
/**
   Constructor with one parameter that sets the last name.
   @param aName a new last name which is assigned to the
   student.
*/
  public Student(String aName)
  {
    lastName = aName;
  }


// return value example:
@return a string containing the student's last name.
```

Using the Javadoc utility - 4

```
/**
 * A class for simulating an integer memory cell
 * @author Mark A. Weiss
 */
public class IntCell
{   /** Get the stored value.
     * @return the stored value.
     */
    public int read( )  {
        return storedValue;
    }
    /** Store a value
     * @param x the number to store.
     */
    public void write( int x ) {
        storedValue = x;
    }

    private int storedValue;
}
```

Figure 3.4, page 66

Figure 3.5 (A)

javadoc output for Figure 3.4 (partial output) (*continued*)

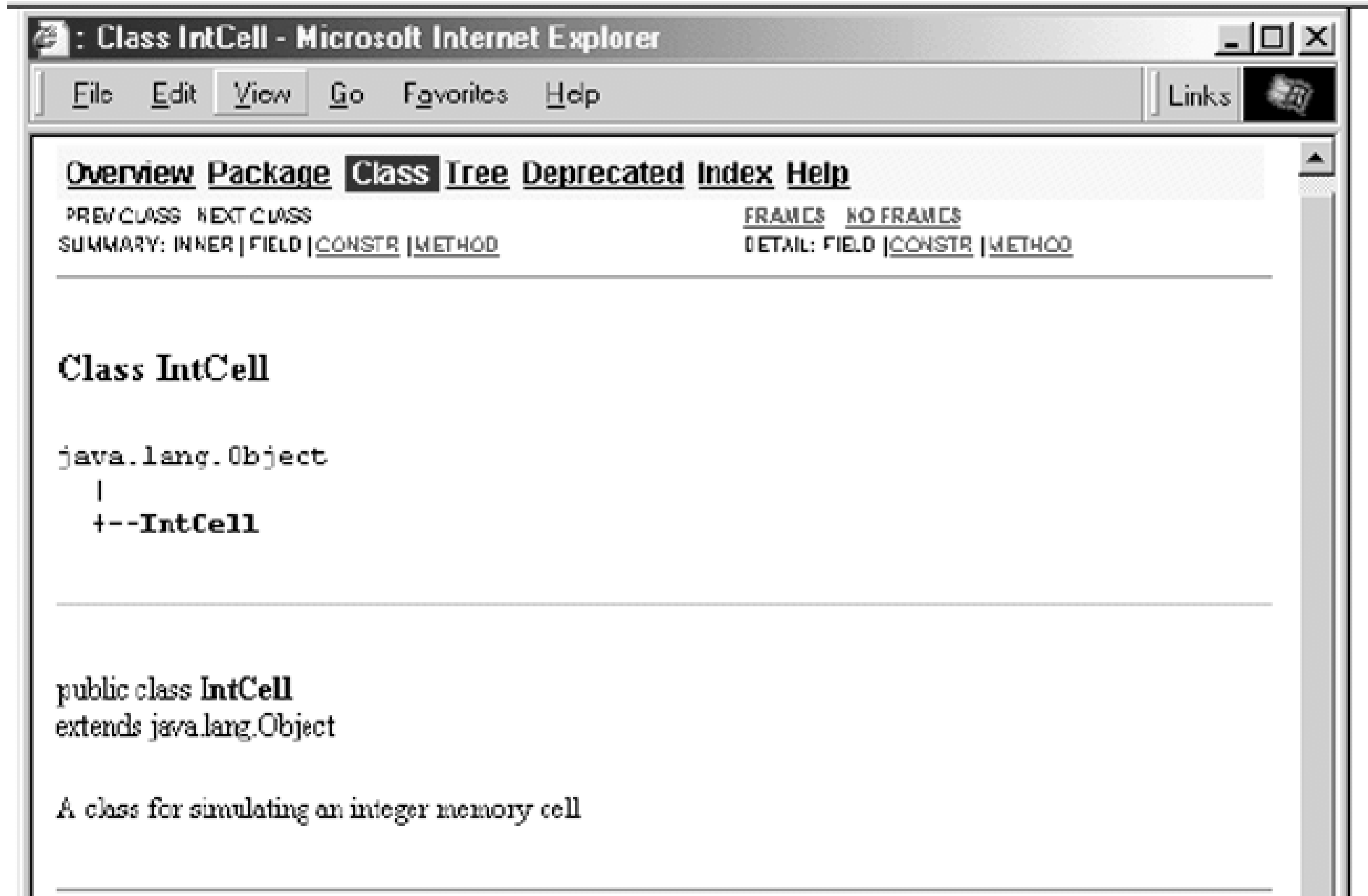

Figure 3.5 (B)
javadoc output for Figure 3.4 (partial output)

Constructor Summary

`IntCell()`

Method Summary

int	`read()` Get the stored value.
void	`write(int x)` Store a value

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

© 2002 Addison Wesley

Inheritance

- Defines a IS-A relationship between classes.
- Base classes and derived classes.
- Derived class inherits all fields and methods of base class.
- Derived class objects are type compatible with base class.
- protected fields and methods: visible to derived classes and to classes in same package.
- inheritance is transitive.
- polymorphism allows for redefining fields and methods.
- dynamic binding allows for run-time determination of overloads and/or overrides.
- super() is a way to refer to constructor of base class.
  It can also be called using appropriate parameters.
  It can only be the first line of a constructor.
- super with appropriate parameters is also used to invoke the corresponding method of the base class.

```
class Person // Fig 4.1, page 91
{
   public Person( String n, int ag, String ad, String p )
   {  name = n; age = ag; address = ad; phone = p;  }

   public String toString( )
   {return getName( ) + " " + getAge( ) + " " + getPhoneNumber( );  }

   public final String getName( )
   {   return name;  }

   public final int getAge( )
   {  return age;  }

   public final String getAddress( )
   {  return address;  }

   public final String getPhoneNumber( )
   {  return phone;  }

   public final void setAddress( String newAddress )
   {  address = newAddress;  }

   public final void setPhoneNumber( String newPhone )
   {  phone = newPhone; }

   private String name;
   private int age;
   private String address;
   private String phone;
}
```

```
class Student extends Person // Fig 4.8, page 102
{
   public Student( String n, int ag, String ad, String p, double g )
   {
      super( n, ag, ad, p );
      gpa = g;
   }

   public String toString( )
   {
      return super.toString( ) + " " + getGPA();
   }

   public double getGPA( )
   {
      return gpa;
   }

   private double gpa;
}
```

```
class PersonDemo // Fig 4.9, pg 103
{
    public static void printAll( Person[ ] arr )
    {
        for( int i = 0; i < arr.length; i++ )
        {
            if( arr[ i ] != null )
            {
                System.out.print( "[" + i + "] " + arr[ i ] );
                System.out.println( );
            }
        }
    }

    public static void main( String [ ] args )
    {
        Person [ ] p = new Person[ 4 ];
        p[0] = new Person( "joe", 25, "New York", "212-555-1212" );
        p[1] = new Student( "becky", 27, "Chicago", "312-555-1212", 4.0 );
        p[3] = new Employee( "bob", 29, "Boston", "617-555-1212", 100000.0 );

        if( p[3] instanceof Employee )
            ((Employee) p[3]).raise( .04 );

        printAll( p );
    }
}
```

Abstract Methods & Classes

- abstract methods are not implemented (not even a default one).
- This is better than putting in a dummy procedure as a placeholder.
- Derived classes must eventually implement them;
  if they don't then they must be abstract classes themselves.
- Overriding is resolved at runtime.
- Abstract class is one that contains an abstract method;
  need to be explicitly declared as such.
- Abstract classes may have non-abstract methods & static fields.
- Abstract classes cannot be created (no constructor),
  except using `super()`

```java
public abstract class Shape
{
   public abstract double area( );
   public abstract double perimeter( );

   public double semiperimeter( )
   {  return perimeter( ) / 2; }
}
```

```java
class ShapeDemo // Fig 4.11 & 4.12, pg 104-5
{
   public static double totalArea( Shape [ ] arr )
   {
      double total = 0;

      for( int i = 0; i < arr.length; i++ )
      {
         if( arr[ i ] != null )
            total += arr[ i ].area( );
      }

      return total;
   }

public static void printAll( Shape [ ] arr )
   {
      for( int i = 0; i < arr.length; i++ )
         System.out.println( arr[ i ] );
   }

   public static void main( String [ ] args )
   {
      Shape [ ] a = { new Circle( 2.0 ), new Rectangle( 1.0, 3.0 ),
                    null, new Square( 2.0 ) };
      System.out.println( "Total area = " + totalArea( a ) );
      System.out.println( "Total semiperimeter = " +
                  totalSemiperimeter( a ) );
      printAll( a );
   }
}
```

Multiple Inheritance using interface

- An interface is an "ultimate" abstract class;
- no implementations are allowed.
- A class may extend only one other base class, but may implement multiple interfaces (thus avoiding conflicting multiple inheritances).
- All methods specified in the interface must be implemented.
- If not, it must be declared "abstract".
- All interfaces & their implementations are "public".
- Interfaces can extend other interfaces.
- Interfaces aren't classes; you can't construct interface objects; you can create variables whose type is an interface, but it will not point to an interface object.

```
Package java.lang;

// Figs 4.15 & 4.16, pg 110-1

public interface Comparable
{ //automatically public
   int compareTo( Object other );
}
```

Also read pages 363-373, Big Java.

```
public abstract class Shape implements Comparable
{
   public abstract double area( );
   public abstract double perimeter( );

   public int compareTo( Object rhs )
   { // must be declared public
      Shape other = (Shape) rhs;
      double diff = area( ) - other.area( );
      if( diff == 0 )
         return 0;
      else if( diff < 0 )
         return -1;
      else
         return 1;
   }

   public double semiperimeter( )
   {
      return perimeter( ) / 2;
   }
}
```

Generic Implementations

- If the implementation is identical except for the basic type, then `Object` type is used to get generic implementations.
- This is the equivalent of "template" in C++; every reference type is compatible with the `Object` type.
- When specific methods of the object are needed, then we need to "downcast" to the correct type.
- If a class does not extend another class, it extends the class Object. It is a class (not abstract) with several methods including toString().

- If a method required is not available in Object, then generic implementations can be achieved using interface.

```
// MemoryCell class // Fig 4.21 & 4.22, pg 119
//  Object read( )        -->  Returns the stored value
//  void write( Object x ) -->  x is stored

public class MemoryCell
{
        // Public methods
    public Object read( )         { return storedValue; }
    public void write( Object x ) { storedValue = x; }

        // Private internal data representation
    private Object storedValue;
}
```

```
public class TestMemoryCell
{
    public static void main( String [ ] args )
    {
        MemoryCell m = new MemoryCell( );

        m.write( "57" );
        String val = (String) m.read( );
        System.out.println( "Contents are: " + val );
    }
}
```

```
class FindMaxDemo // Fig 4.26, pg 123
{

    /**
     * Return max item in a.
     * Precondition: a.length > 0
     */
    public static Comparable findMax( Comparable [ ] a )
    {
        int maxIndex = 0;

        for( int i = 1; i < a.length; i++ )
            if( a[ i ].compareTo( a[ maxIndex ] ) > 0 )
                maxIndex = i;

        return a[ maxIndex ];
    }
```

```
/**
     * Test findMax on Shape and String objects.
     * Shape implements "Comparable"
     */
    public static void main( String [ ] args )
    {
        Shape [ ] sh1 = { new Circle(  2.0 ),
                          new Square(  3.0 ),
                          new Rectangle( 3.0, 4.0 ) };

        String [ ] st1 = { "Joe", "Bob", "Bill", "Zeke" };

        System.out.println( findMax( sh1 ) );
        System.out.println( findMax( st1 ) );
    }
}
```

Functors

- A functor is an object with no data and a single method.
- Functors can be passed as parameters.
- Since these classes are very "small", they are usually implemented as a Nested Class wherever they are needed.
- Nested classes are defined inside other classes and it is essential that it be declared as "static". If it is not declared as "static", then it is an "inner" class (not nested).
- Nested classes act as members of the "outer" class, and can be declared as private, public, protected, or package visible.
- A nested class can access private fields and members of the "outer" class.
- Functors can also be implemented as a Local Class or as an Anonymous Class.

Functors

```java
// Fig 4.29 & 4.30, pg 127

import java.util.Comparator;

class OrderRectByArea implements Comparator
{
    public int compare( Object obj1, Object obj2 )
    {
        SimpleRectangle r1 = (SimpleRectangle) obj1;
        SimpleRectangle r2 = (SimpleRectangle) obj2;

        return( r1.getWidth()*r1.getLength() -
                r2.getWidth()*r2.getLength() );
    }
}
```

```java
public class CompareTest
{
    public static Object findMax( Object [ ] a,
                                  Comparator cmp )
    {
        int maxIndex = 0;
        for( int i = 1; i < a.length; i++ )
            if( cmp.compare( a[ i ], a[ maxIndex ] ) > 0 )
                maxIndex = i;
        return a[ maxIndex ];
    }

    public static void main( String [ ] args )
    {
        Object [ ] rects = new Object[ 4 ];
        rects[ 0 ] = new SimpleRectangle( 1, 10 );
        rects[ 1 ] = new SimpleRectangle( 20, 1 );
        rects[ 2 ] = new SimpleRectangle( 4, 6 );
        rects[ 3 ] = new SimpleRectangle( 5, 5 );

        System.out.println( "MAX WIDTH: " +
           findMax( rects, new OrderRectByWidth( ) ) );
        System.out.println( "MAX AREA: " +
           findMax( rects, new OrderRectByArea( ) ) );
    }
}
```

Nested Classes

```
import java.util.Comparator;
// Fig 4.32 pg 130
class CompareTestInner1
{
    public static Object findMax( Object [ ] a,
                                  Comparator cmp )
    {
        int maxIndex = 0;
        for( int i = 1; i < a.length; i++ )
            if( cmp.compare( a[ i ], a[ maxIndex ] ) > 0 )
                maxIndex = i;

        return a[ maxIndex ];
    }

private static class OrderRectByArea
            implements Comparator
    {
        public int compare( Object obj1, Object obj2 )
        {
            SimpleRectangle r1 = (SimpleRectangle) obj1;
            SimpleRectangle r2 = (SimpleRectangle) obj2;

            return( r1.getWidth()*r1.getLength() -
                    r2.getWidth()*r2.getLength() );
        }
    }
```

```
public static void main( String [ ] args )
    {
        Object [ ] rects = new Object[ 4 ];
        rects[ 0 ] = new SimpleRectangle( 1, 10 );
        rects[ 1 ] = new SimpleRectangle( 20, 1 );
        rects[ 2 ] = new SimpleRectangle( 4, 6 );
        rects[ 3 ] = new SimpleRectangle( 5, 5 );

        System.out.println( "MAX WIDTH: " +
            findMax( rects, new OrderRectByWidth( ) ) );
        System.out.println( "MAX AREA: " +
            findMax( rects, new OrderRectByArea( ) ) );
    }
}
```

Local Classes

```
import java.util.Comparator;
// Fig 4.33 pg 131
class CompareTestInner2
{
   public static Object findMax( Object [ ] a,
              Comparator cmp )
   {
      int maxIndex = 0;
      for( int i = 1; i < a.length; i++ )
         if( cmp.compare( a[ i ], a[ maxIndex ] ) > 0 )
            maxIndex = i;

      return a[ maxIndex ];
   }
```

```
public static void main( String [ ] args )
   {  Object [ ] rects = new Object[ 4 ];
      rects[ 0 ] = new SimpleRectangle( 1, 10 );
      rects[ 1 ] = new SimpleRectangle( 20, 1 );
      rects[ 2 ] = new SimpleRectangle( 4, 6 );
      rects[ 3 ] = new SimpleRectangle( 5, 5 );

  // neither public nor static
  class OrderRectByArea implements Comparator
      {
         public int compare( Object obj1, Object obj2 )
         {
            SimpleRectangle r1 = (SimpleRectangle) obj1;
            SimpleRectangle r2 = (SimpleRectangle) obj2;

            return( r1.getWidth()*r1.getLength() -
                  r2.getWidth()*r2.getLength() );
         }
      }

   System.out.println( "MAX AREA: " +
            findMax( rects, new OrderRectByArea( ) ) );
   }
}
```

Anonymous Classes

```
class CompareTestInner3 // Fig 4.34, pg 132
{
public static void main( String [ ] args )
    {
        Object [ ] rects = new Object[ 4 ];
        rects[ 0 ] = new SimpleRectangle( 1, 10 );
        rects[ 1 ] = new SimpleRectangle( 20, 1 );
        rects[ 2 ] = new SimpleRectangle( 4, 6 );
        rects[ 3 ] = new SimpleRectangle( 5, 5 );
        System.out.println( "MAX WIDTH: " + findMax( rects, new Comparator( )
            {  // no name class, no constructor
                public int compare( Object obj1, Object obj2 )
                {
                    SimpleRectangle r1 = (SimpleRectangle) obj1;
                    SimpleRectangle r2 = (SimpleRectangle) obj2;
                    return( r1.getWidth() - r2.getWidth() );
                }
            }
        ) );
        System.out.println( "MAX AREA: " + findMax( rects, new Comparator( )
            {
                public int compare( Object obj1, Object obj2 )
                {
                    SimpleRectangle r1 = (SimpleRectangle) obj1;
                    SimpleRectangle r2 = (SimpleRectangle) obj2;
                    return( r1.getWidth()*r1.getLength() -  r2.getWidth()*r2.getLength() );
                }
            }
        ) );
    }
}
```

Overriding vs. Overloading

- In a derived class, if a method declaration does not match the exact signature, then it is not an "override", but an "overload".
- If a method is declared as final, it cannot be overridden.
- If a class is declared as final, it cannot be extended.

Packages

- Group of related classes.
- Specified by package statement.
- Fewer restrictions on access among each other;
  - if class is called public, then it is visible to all classes
  - if no visibility modifier is specified, it is equivalent to the friend specification from C++, and its visibility is termed as "package visibility" and is somewhere between:
    - private (other classes in package cannot access it) and
    - public (other classes outside package can also access it)
  - A class cannot be private or protected. Only methods & fields are allowed to be declared as such.
- Package locations can be specified by the CLASSPATH environmental variables.
- The import statement helps to get multiple packages. It saves typing.

Access Restrictions of Methods/Fields

- Clients have access to only public methods.
- Derived classes have access to public & protected members of the base class.
- Classes within the same package have access to protected and package members of the base class.

- Public – can be used by anyone .
- Package – by methods of the class and in same package.
- Protected – by methods of the class and subclasses and in the same package.
- Private – only by members of the same class.

```
public final class MaxSumTest
{  // Fig 5.4, p155
    static private int seqStart = 0;
    static private int seqEnd = -1;
    public static int maxSubSum1( int [ ] a )
    {
        int maxSum = 0;

        for( int i = 0; i < a.length; i++ )
            for( int j = i; j < a.length; j++ )
            {
                int thisSum = 0;

                for( int k = i; k <= j; k++ )
                    thisSum += a[ k ];

                if( thisSum > maxSum )
                {
                    maxSum   = thisSum;
                    seqStart = i;
                    seqEnd   = j;
                }
            }

        return maxSum;
    }
}
```

```
public final class MaxSumTest
{  // Fig 5.5, p157
    public static int maxSubSum2( int [ ] a )
    {
        int maxSum = 0;

        for( int i = 0; i < a.length; i++ )
        {
            int thisSum = 0;
            for( int j = i; j < a.length; j++ )
            {
                thisSum += a[ j ];

                if( thisSum > maxSum )
                {
                    maxSum = thisSum;
                    seqStart = i;
                    seqEnd   = j;
                }
            }
        }

        return maxSum;
    }
}
```

```
public final class MaxSumTest
{  // Fig 5.8, p160
   public static int maxSubSum3( int [ ] a )
   {
      int maxSum = 0;
      int thisSum = 0;

      for( int i = 0, j = 0; j < a.length; j++ )
      {
         thisSum += a[ j ];

         if( thisSum > maxSum )
         {
            maxSum = thisSum;
            seqStart = i;
            seqEnd   = j;
         }
         else if( thisSum < 0 )
         {
            i = j + 1;
            thisSum = 0;
         }
      }

      return maxSum;
   }
}
```

Containers

- Powerful tool for programming data structures
- Provides a library of container classes to "hold your objects"
- 2 types of Containers:
  - Collection: to hold a group of elements e.g., List, Set
  - Map: a group of key-value object pairs. It helps to return "Set of keys, collection of values, set of pairs. Also works with multiple dimensions (i.e., map of maps).
- Iterators give you a better handle on containers and helps to iterate through all the elements. It can be used without any knowledge of how the collection is implemented.
- Collections API provides a few general purpose algorithms that operate on all containers.

```java
// Fig 6.9, 6.10, pg 192, 194.
package weiss.util;

public interface Collection extends java.io.Serializable
{
   int size( );
   boolean isEmpty( );
   boolean contains( Object x );
   boolean add( Object x );
   boolean remove( Object x );
   void clear( );
   Iterator iterator( );
   Object [ ] toArray( );
}

public interface Iterator
{
   boolean hasNext( );
   Object next( );
   void remove( );
}
```

```java
// Fig 6.11, pg 195
public static void printCollection
            (Collection c)
{
   Iterator itr = c.iterator();
   while (itr.hasNext())
      System.out.println(itr.next());
}
```