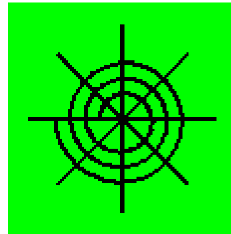
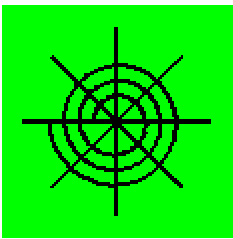
Java Interfaces – Part 2
The Vector Class, Sorting, and Collections



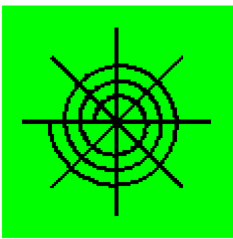
By
Dr. Nicholas Duchon
July 5, 2007

Version 1.1



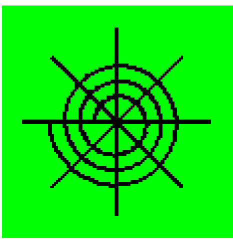
Overview

- Introduction:
  - Overview, Relation to Interfaces, and History,
- Simple array sorting
- The Comparable and Comparator interfaces
  - Sorting arrays of user defined classes
- The old Vector class and its limitations
- Collections classes
- Generic data structures – as we go along



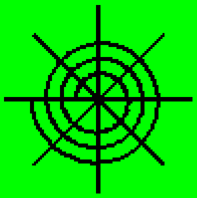
Relation to Interfaces

- Why are we looking at data structures in a series of lectures on interfaces?
  - It turns out that two of the most interesting interfaces are associated with data structures: Comparable and Comparator.
  - These interfaces are used by the collections classes in sorting algorithms.
  - These interfaces are "generic" in the Java sense of the word generic, so we need to understand what that means and at least some of the implications.
  - And just for fun, Java has rearranged the class hierarchy of some of the original data structure classes, and made those classes generic also.
- As with other interfaces, if we program according to Java's expectations, we can USE the classes and interfaces Java has already defined for us to create very compact, exceptionally powerful, and truly elegant programs.



History of Java Collections Classes

- A bit of history
  - Java started with a very few data structures in the JDK version 1.0: basically array of primitive data types and Objects, and the interesting data class Vector.
  - Those who have studied data structures will immediately recognize that using only these two structures seems rather limiting. Since then, Java has added a whole set of classes, implementing the Collection interface, which include a variety of interesting structures, starting in JDK 2.0: Set, SortedSet, List, Queue, Map and SortedMap.
  - JDK 3 and 4 introduced more data structure classes.
  - JDK 5 introduced generics, which added syntax to the compiler. This is a most dramatic departure from previous releases, but we will address the issues raised by generics as we go along.

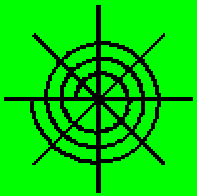


Array of int

- Vector – primary methods
  - add
  - get
  - indexOf
  - remove, clear
  - size
  - toString

```
import java.util.Vector;
import java.awt.Point;

class MyVectorDemoA {
     public static void main (String args []) {
          Vector v = new Vector ();
          v.add (new String ("this is the first"));
          v.add (new Point (12, 33));
          v.add (new String ("the third entry"));
          System.out.println ("The vector:\n" + v);
     } // end main
} // end class MyVectorDemoA

// Output:
/*
 The vector:
[this is the first, java.awt.Point[x=12,y=33], the third entry]
*/
```

This is a very simple example that demonstrates the flexibility of the Vector class.
Notice that this example puts objects of type String and Point into the same Vector.
The 5.0 compiler gives warnings about unchecked calls to "add", but this code works nicely.
Notice also the implicit use of toString in println, which then calls toString on each element in the Vector.



Sorting Arrays.sort

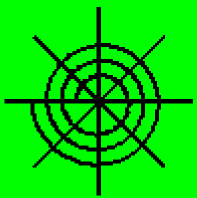
- A useful method to print an array of int's.
- Creating an array of random int's – useful for sorting demonstrations.
- The INTERESTING line of code!

```
import java.util.Random;
import java.util.Arrays;

public class MySortA {
  public static void printArray (int a []) {
    for (int i = 0; i < a.length; i++) {
      if (i % 10 == 0) System.out.println ();
      System.out.printf ("%6d", a[i]);
    } // end for each element in array
  } // end printArray

  public static void main (String args []) {
    Random rn = new Random ();
    int x [] = new int [60];
    for (int i = 0; i < x.length; i++)
      x[i] = rn.nextInt (10000);

    printArray (x);

    Arrays.sort (x);

    System.out.println ("\n\nSorted:");
    printArray (x);
  } // end main
} // end class MySortA
```

In this example, we see an array of 60 int's created and initialized with random numbers less than 10,000, sorted using a JDK static method: Arrays.sort (int []).

This code also gives a nice general purpose method for printing an array.

The thing to notice in the program is how simple the call to the sort routine provided by the JDK is, and the simplicity of the code in main.



Sorting A Arrays.sort

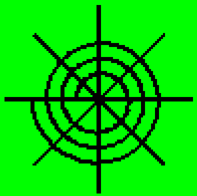
- A useful method to print an array of int's.
- Creating an array of random int's – useful for sorting demonstrations.
- The INTERESTING line of code!

```
import java.util.Random;
import java.util.Arrays;

public class MySortA {
  public static void printArray (int a []) {
    for (int i = 0; i < a.length; i++) {
      if (i % 10 == 0) System.out.println ();
      System.out.printf ("%6d", a[i]);
    } // end for each element in array
  } // end printArray

  public static void main (String args []) {
    Random rn = new Random ();
    int x [] = new int [60];
    for (int i = 0; i < x.length; i++)
      x[i] = rn.nextInt (10000);

    printArray (x);

    Arrays.sort (x);

    System.out.println ("\n\nSorted:");
    printArray (x);
  } // end main
} // end class MySortA
```

In this example, we see an array of 60 int's created and initialized with random numbers less than 10,000, sorted using a JDK static method: Arrays.sort (int []).

This code also gives a nice general purpose method for printing an array.

The thing to notice in the program is how simple the call to the sort routine provided by the JDK is, and the simplicity of the code in main.



Sorting B Strings

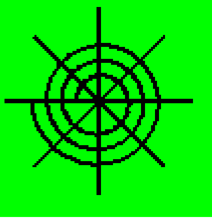
- Only one instance of Random
- A useful method to print an array of String's.
- Creating an array of random len-character Strings – again useful for sorting demonstrations.
- The INTERESTING line of code! Using the natural order of the String class.

The next example is a little more complicate - it uses an array of String rather than int.

We have moved the creation of the array into its own method, which lets the main method be very simple:
•Create an array
•Print it
•Sort it
•Print the sorted one

The printArray method is a little more complicated than the one for ints, but not so very much.

```
import java.util.Random;
import java.util.Arrays;

public class MySortB {
  static Random rn = new Random ();

  public static void printArray (String a []) {
    for (int i = 0; i < a.length; i++) {
      if (i % 10 == 0) System.out.println ();
      System.out.printf ("%" + (a[i].length() + 1) + "s", a[i]);
    } // end for each element in array
  } // end printArray

  public static String [] randomStrings (int num, int len) {
    String x [] = new String [num];
    char c [] = new char [len];

    for (int i = 0; i < x.length; i++) {
     for (int j = 0; j < c.length; j++)
        c[j] = (char) ('a' + rn.nextInt (26));
      x[i] = new String (c);
   } // end for each array element

    return x;
  } // end method randomStrings

  public static void main (String args []) {
    String x [] = randomStrings (50, 4);
    printArray (x);
    Arrays.sort (x);
    System.out.println ("\n\nSorted:");
    printArray (x);
  } // end main
} // end class MySortB
```



Sorting B Strings

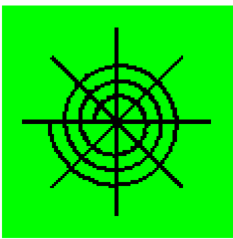
- Only one instance of Random
- A useful method to print an array of String's.
- Creating an array of random len-character Strings – again useful for sorting demonstrations.
- The INTERESTING line of code! Using the natural order of the String class.

The next example is a little more complicate - it uses an array of String rather than int.

We have moved the creation of the array into its own method, which lets the main method be very simple:
•Create an array
•Print it
•Sort it
•Print the sorted one

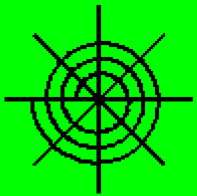
The printArray method is a little more complicated than the one for ints, but not so very much.

```
import java.util.Random;
import java.util.Arrays;

public class MySortB {
  static Random rn = new Random ();

  public static void printArray (String a []) {
    for (int i = 0; i < a.length; i++) {
      if (i % 10 == 0) System.out.println ();
      System.out.printf ("%" + (a[i].length() + 1) + "s", a[i]);
    } // end for each element in array
  } // end printArray

  public static String [] randomStrings (int num, int len) {
    String x [] = new String [num];
    char c [] = new char [len];

    for (int i = 0; i < x.length; i++) {
     for (int j = 0; j < c.length; j++)
       c[j] = (char) ('a' + rn.nextInt (26));
      x[i] = new String (c);
   } // end for each array element

    return x;
  } // end method randomStrings

  public static void main (String args []) {
    String x [] = randomStrings (50, 4);
    printArray (x);
    Arrays.sort (x);
    System.out.println ("\n\nSorted:");
    printArray (x);
  } // end main
} // end class MySortB
```



Sorting Order?

- How does Arrays.sort (int []) or Arrays.sort (Object []) know how to order the elements in the array?
- Hint: the array in the second version is an array of Object, but each element of the array must implement the interface Comparable, otherwise a run-time exception will be thrown.
- The String class implements Comparable, the instance method `int compareTo (Object)`
- The method compareTo should satisfy the following properties, otherwise very bad things can happen:
  - If a and b are in the array, then a.compareTo(b) and b.compareTo (a) must be OK.
  - a.compareTo(b) < 0 if and only if b.compareTo(a) > 0.
  - If a.compareTo(b) = 0, then b.compareTo(a) = 0.
- The next examples will show how to use the Comparable interface and Arrays.sort to sort in orders other than the defaults.

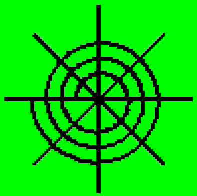


Sorting C MyString

Sorting an array of String, using a class that has a String as an instance variable, and implements the Comparable interface.

- Imports, Random, printArray as in the previous example.
- randomStrings similar, but returns an array of MyString
- main is also the same, except for using MyString
- The new class MyString.
  - String instance variable
  - char [] constructor
  - length and toString defined to work the same way as the String class
  - compareTo (Object) is the reverse (negative) of the method defined in String

```
// imports as before
   public class MySortC {
      static Random rn = new Random ();

      public static void printArray (MyString) {...}

      public static MyString [] randomStrings (int, int) {
         MyString x [] = new MyString [num];
         char c [] = new char [len];

         for (int i = 0; i < x.length; i++) {
            for (int j = 0; j < c.length; j++)
               c[j] = (char) ('a' + rn.nextInt (26));
            x[i] = new MyString (c);
         } // end for each array element

      public static void main (String args []) {
         MyString x [] = randomStrings (50, 4);
         printArray (x);
         Arrays.sort (x); // x must implement Comparable
         System.out.println ("\n\nSorted, reverse order:");
         printArray (x);
      } // end main

   } // end class MySortC

   class MyString implements Comparable {
      String st;
      public MyString (char [] ca) {
         st = new String (ca);
      } // end char[] constructor
      public int    length   () {return st.length();}
      public String toString () {return st;}

      public int compareTo (Object ob) {
         if (! (ob instanceof MyString)) return 0;
         return - st.compareTo (((MyString)ob).st);
      } // end compareTo
   } // end MyString
```



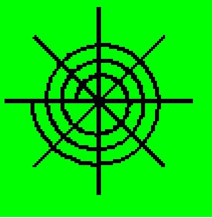
Sorting C MyString

Sorting an array of String, using a class that has a String as an instance variable, and implements the Comparable interface.

- Imports, Random, printArray as in the previous example.
- randomStrings similar, but returns an array of MyString
- main is also the same, except for using MyString
- The new class MyString.
  - String instance variable
  - char [] constructor
  - length and toString defined to work the same way as the String class
  - compareTo (Object) is the reverse (negative) of the method defined in String

```
// imports as before
   public class MySortC {
      static Random rn = new Random ();

      public static void printArray (MyString) {...}

      public static MyString [] randomStrings (int, int) {
         MyString x [] = new MyString [num];
         char c [] = new char [len];

         for (int i = 0; i < x.length; i++) {
            for (int j = 0; j < c.length; j++)
               c[j] = (char) ('a' + rn.nextInt (26));
            x[i] = new MyString (c);
         } // end for each array element

      public static void main (String args []) {
         MyString x [] = randomStrings (50, 4);
         printArray (x);
         Arrays.sort (x); // x must implement Comparable
         System.out.println ("\n\nSorted, reverse order:");
         printArray (x);
      } // end main

   } // end class MySortC

   class MyString implements Comparable {
      String st;
      public MyString (char [] ca) {
         st = new String (ca);
      } // end char[] constructor
      public int    length   () {return st.length();}
      public String toString () {return st;}

      public int compareTo (Object ob) {
         if (! (ob instanceof MyString)) return 0;
         return - st.compareTo (((MyString)ob).st);
      } // end compareTo
   } // end MyString
```

Negative sign!

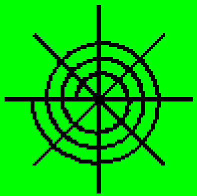


Sorting C MyString

Assessment:

- The code in main is nearly identical to MySortB, which is nice!
- Creating a new class that is almost identical to String is NOT so nice. The problem is that String is a final class and cannot simply be extended. However, we will see in MySortG a nice way around this limitation.
- Because MyString does not extend String, methods that were used in MySortB from the String class needed to be explicitly defined (char [] constructor, length and toString).
- The Comparable interface requires the compareTo (Object) method. This code USES String.compareTo (Object).
  Notice the casting required.
- The MyString class still only has ONE ordering – the next examples will show classes with more than one ordering.

```
// imports as before
   public class MySortC {
      static Random rn = new Random ();

      public static void printArray (MyString) {...}

      public static MyString [] randomStrings (int, int) {
         MyString x [] = new MyString [num];
         char c [] = new char [len];

         for (int i = 0; i < x.length; i++) {
            for (int j = 0; j < c.length; j++)
               c[j] = (char) ('a' + rn.nextInt (26));
            x[i] = new MyString (c);
         } // end for each array element

      public static void main (String args []) {
         MyString x [] = randomStrings (50, 4);
         printArray (x);
         Arrays.sort (x); // x must implement Comparable
         System.out.println ("\n\nSorted, reverse order:");
         printArray (x);
      } // end main

   } // end class MySortC

   class MyString implements Comparable {
      String st;
      public MyString (char [] ca) {
         st = new String (ca);
      } // end char[] constructor
      public int    length   () {return st.length();}
      public String toString () {return st;}

      public int compareTo (Object ob) {
         if (! (ob instanceof MyString)) return 0;
         return - st.compareTo (((MyString)ob).st);
      } // end compareTo
   } // end MyString
```

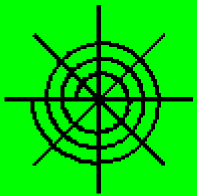


Sorting D 2D int []

This example shows some examples of working with 2-D arrays.

- main is pretty much the same program we have already seen: init the array, print it, sort it, and print it again.
- getArray creates a 2-D array of random int's. The parameters allow it to be used to create arrays of various sizes and ranges of elements.
- printArray allows the caller to specify the width used to display each int. Note the use of the nested advanced for loops.
- sortRows is the simplest approach to sorting, each row is sorted from smallest to largest using the default order of int's, and the built-in Arrays.sort (int []) method.

```
// Sorting each row in a 2-D int array

import java.util.Random;
import java.util.Arrays;

public class MySortD {
  public static Random rn = new Random ();

  public static void main (String args []) {
    int [][] b = getArray (15, 10, 100);
    printArray (b, 4);
    sortRows (b);
    System.out.println ("Sorted:");
    printArray (b, 4);
  } // end main

  static int [][] getArray (int r, int c, int m) {
    int [][] arv = new int [r][c];
    for (int i = 0; i < r; i++)
      for (int j = 0; j < c; j++)
        arv[i][j] = rn.nextInt (m);
    return arv;
  } // end create random array

  static void printArray (int [][] arp, int w) {
    for (int x [] : arp) {
      for (int y : x)
        System.out.printf ("%" + w + "d", y);
     System.out.println ();
    } // end for each row
  } // end printArray

  static void sortRows (int [][] arp) {
    for (int x [] : arp)
      Arrays.sort (x);
  } // end sortRows

} // end class MySortD
```

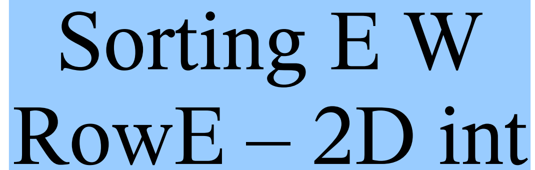


Sorting D
2D int []

This example shows some examples of working with 2-D arrays.

- main is pretty much the same program we have already seen: init the array, print it, sort it, and print it again.
- getArray creates a 2-D array of random int's. The parameters allow it to be used to create arrays of various sizes and ranges of elements.
- printArray allows the caller to specify the width used to display each int. Note the use of the nested advanced for loops.
- sortRows is the simplest approach to sorting, each row is sorted from smallest to largest using the default order of int's, and the built-in Arrays.sort (int []) method.

```
// Sorting each row in a 2-D int array

import java.util.Random;
import java.util.Arrays;

public class MySortD {
  public static Random rn = new Random ();

  public static void main (String args []) {
    int [][] b = getArray (15, 10, 100);
    printArray (b, 4);
    sortRows (b);
    System.out.println ("Sorted:");
    printArray (b, 4);
  } // end main

  static int [][] getArray (int r, int c, int m) {
    int [][] arv = new int [r][c];
    for (int i = 0; i < r; i++)
      for (int j = 0; j < c; j++)
        arv[i][j] = rn.nextInt (m);
    return arv;
  } // end create random array

  static void printArray (int [][] arp, int w) {
    for (int x [] : arp) {
      for (int y : x)
        System.out.printf ("%" + w + "d", y);
     System.out.println ();
    } // end for each row
  } // end printArray

  static void sortRows (int [][] arp) {
    for (int x [] : arp)
      Arrays.sort (x);
  } // end sortRows

} // end class MySortD
```



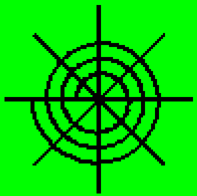
Sorting E W RowE – 2D int

```
// new class Rows to demonstrate Comparable
// interface made effective use of Arrays.sort
// using Comparable
// NOT thread safe, or multiple instance safe

import java.util.Random;
import java.util.Arrays;

public class MySortE {
  public static void main (String args []) {
    RowE [] rr = newArray (15, 5, 100);
    printArray (rr);

    RowE.setCol  (3);      // sort by column 3
    RowE.setUp   (false); // sort descending
    Arrays.sort (rr);

    System.out.println ("--- Sorted:");
    printArray (rr);
    // works, but NOT pretty!
    // System.out.println (Arrays.toString (rr));
  } // end main

  static RowE [] newArray (int r, int c, int m) {
    RowE [] arr = new RowE [r];
    // entries null at this point
    for (int i = 0; i < r; i++)
      arr [i] = new RowE (c, m);
    return arr;
  } // end newArray

  static void printArray (RowE [] a) {
    for (RowE r : a)
      System.out.println (r);
  } // end printArray
} // end class MySortE
```

```
class RowE implements Comparable {
  int [] row; // starts as null

  static int sortColumn = 0;
  static boolean orderAZ = true;
  static int printWidth = 5;
  static Random rn = new Random ();

  public RowE (int c, int m) {
    row = new int [c];
    for (int i = 0; i < c; i++)
      row[i] = rn.nextInt (m);
  } // end random constructor

  public String toString () {
    String st = "";
    for (int x : row)
      st += String.format ("%" + printWidth + "d", x);
    return st;
  } // end toString

  public static void setCol  (int     v) {sortColumn = v;}
  public static void setUp   (boolean t) {orderAZ = t;}

 // instance method required by Comparable
  public int compareTo (Object o) {
    if (! (o instanceof RowE)) return 0;
    RowE r = (RowE) o;
    if (orderAZ) return row[sortColumn] - r.row[sortColumn];
    return r.row[sortColumn] - row[sortColumn];
  } // end method compareTo
} // end RowE
```

This example sorts the rows of the 2-D array of ints (well, sort of), by creating a custom class, RowE, for each row.

RowE uses the compareTo method, with two flags: sortColumn to select which column to sort, and orderAZ to pick A to Z vs Z to A ordering.



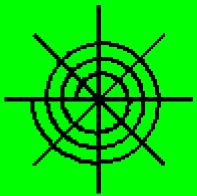
Sorting E W RowE – 2D int

```java
// new class Rows to demonstrate Comparable
// interface made effective use of Arrays.sort
// using Comparable
// NOT thread safe, or multiple instance safe

import java.util.Random;
import java.util.Arrays;

public class MySortE {
  public static void main (String args []) {
    RowE [] rr = newArray (15, 5, 100);
    printArray (rr);

    RowE.setCol (3);      // sort by column 3
    RowE.setUp  (false); // sort descending
    Arrays.sort (rr);

    System.out.println ("--- Sorted:");
    printArray (rr);
    // works, but NOT pretty!
    // System.out.println (Arrays.toString (rr));
  } // end main

  static RowE [] newArray (int r, int c, int m) {
    RowE [] arr = new RowE [r];
    // entries null at this point
    for (int i = 0; i < r; i++)
      arr [i] = new RowE (c, m);
    return arr;
  } // end newArray

  static void printArray (RowE [] a) {
    for (RowE r : a)
      System.out.println (r);
  } // end printArray
} // end class MySortE
```

```java
class RowE implements Comparable {
  int [] row; // starts as null

  static int sortColumn = 0;
  static boolean orderAZ = true;
  static int printWidth = 5;
  static Random rn = new Random ();

  public RowE (int c, int m) {
    row = new int [c];
    for (int i = 0; i < c; i++)
      row[i] = rn.nextInt (m);
  } // end random constructor

  public String toString () {
    String st = "";
    for (int x : row)
      st += String.format ("%" + printWidth + "d", x);
    return st;
  } // end toString

  public static void setCol (int     v) {sortColumn = v;}
  public static void setUp  (boolean t) {orderAZ = t;}

 // instance method required by Comparable
  public int compareTo (Object o) {
    if (! (o instanceof RowE)) return 0;
    RowE r = (RowE) o;
    if (orderAZ) return row[sortColumn] - r.row[sortColumn];
    return r.row[sortColumn] - row[sortColumn];
  } // end method compareTo
} // end RowE
```

There are two classes:

- MySortE
  - Methods: main, newArray and printArray
- RowE
  - Methodsconstructor, toString, setCol, setUp, compareTo



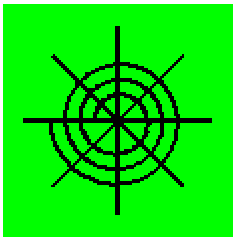
Sorting E W RowE – 2D int

```
// new class Rows to demonstrate Comparable
// interface made effective use of Arrays.sort
// using Comparable
// NOT thread safe, or multiple instance safe

import java.util.Random;
import java.util.Arrays;

public class MySortE {
  public static void main (String args []) {
    RowE [] rr = newArray (15, 5, 100);
    printArray (rr);

    RowE.setCol (3);      // sort by column 3
    RowE.setUp  (false); // sort descending
    Arrays.sort (rr);

    System.out.println ("--- Sorted:");
    printArray (rr);
  } // end main

  static RowE [] newArray (int r, int c, int m) {
    RowE [] arr = new RowE [r];
    // entries null at this point
    for (int i = 0; i < r; i++)
      arr [i] = new RowE (c, m);
    return arr;
  } // end newArray

  static void printArray (RowE [] a) {
    for (RowE r : a)
      System.out.println (r);
  } // end printArray
} // end class MySortE
```

```
class RowE implements Comparable {
  int [] row; // starts as null

  static int sortColumn = 0;
  static boolean orderAZ = true;
  static int printWidth = 5;
  static Random rn = new Random ();

  public RowE (int c, int m) {
    row = new int [c];
    for (int i = 0; i < c; i++)
      row[i] = rn.nextInt (m);
  } // end random constructor

  public String toString () {
    String st = "";
    for (int x : row)
      st += String.format ("%" + printWidth + "d", x);
    return st;
  } // end toString

  public static void setCol (int     v) {sortColumn = v;}
  public static void setUp  (boolean t) {orderAZ = t;}

 // instance method required by Comparable
  public int compareTo (Object o) {
    if (! (o instanceof RowE)) return 0;
    RowE r = (RowE) o;
    if (orderAZ) return row[sortColumn] - r.row[sortColumn];
    return r.row[sortColumn] - row[sortColumn];
  } // end method compareTo
} // end RowE
```

The "well, sort of" comment:

- The array in main is really a 1-D array of elements of type RowE, and
- Each instance of RowE has as its (one) instance variable, a 1-D array of ints.
- The effect is to store a 2-D array of int's, with some additional structure.



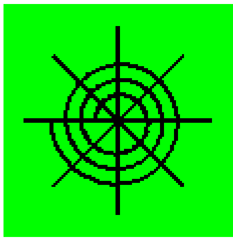
Sorting E W RowE – 2D int

```
// new class Rows to demonstrate Comparable
// interface made effective use of Arrays.sort
// using Comparable
// NOT thread safe, or multiple instance safe

import java.util.Random;
import java.util.Arrays;

public class MySortE {
  public static void main (String args []) {
    RowE [] rr = newArray (15, 5, 100);
    printArray (rr);

    RowE.setCol  (3);      // sort by column 3
    RowE.setUp   (false); // sort descending
    Arrays.sort (rr);

    System.out.println ("--- Sorted:");
    printArray (rr);
  } // end main

  static RowE [] newArray (int r, int c, int m) {
    RowE [] arr = new RowE [r];
    // entries null at this point
    for (int i = 0; i < r; i++)
      arr [i] = new RowE (c, m);
    return arr;
  } // end newArray

  static void printArray (RowE [] a) {
    for (RowE r : a)
      System.out.println (r);
  } // end printArray
} // end class MySortE
```

```
class RowE implements Comparable {
  int [] row; // starts as null

  static int sortColumn = 0;
  static boolean orderAZ = true;
  static int printWidth = 5;
  static Random rn = new Random ();

  public RowE (int c, int m) {
    row = new int [c];
    for (int i = 0; i < c; i++)
      row[i] = rn.nextInt (m);
  } // end random constructor

  public String toString () {
    String st = "";
    for (int x : row)
      st += String.format ("%" + printWidth + "d", x);
    return st;
  } // end toString

  public static void setCol (int     v) {sortColumn = v;}
  public static void setUp  (boolean t) {orderAZ = t;}

 // instance method required by Comparable
  public int compareTo (Object o) {
    if (! (o instanceof RowE)) return 0;
    RowE r = (RowE) o;
    if (orderAZ) return row[sortColumn] - r.row[sortColumn];
    return r.row[sortColumn] - row[sortColumn];
  } // end method compareTo
} // end RowE
```

The algorithm in main is basically the same as the one we have used all along, except that it also explicitly sets the sorting column and direction:

- init the array
- print it
- set the sort criteria (static vars in RowE)
  - column 3 descending
- sort it (Arrays.sort)
- print the sorted array

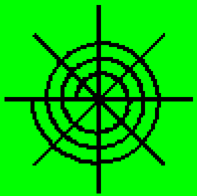


Sorting E W RowE – 2D int

```java
// new class Rows to demonstrate Comparable
// interface made effective use of Arrays.sort
// using Comparable
// NOT thread safe, or multiple instance safe

import java.util.Random;
import java.util.Arrays;

public class MySortE {
  public static void main (String args []) {
    RowE [] rr = newArray (15, 5, 100);
    printArray (rr);

    RowE.setCol  (3);      // sort by column 3
    RowE.setUp   (false); // sort descending
    Arrays.sort (rr);

    System.out.println ("--- Sorted:");
    printArray (rr);
  } // end main

  static RowE [] newArray (int r, int c, int m) {
    RowE [] arr = new RowE [r];
    // entries null at this point
    for (int i = 0; i < r; i++)
      arr [i] = new RowE (c, m);
    return arr;
  } // end newArray

  static void printArray (RowE [] a) {
    for (RowE r : a)
      System.out.println (r);
  } // end printArray
} // end class MySortE
```

```java
class RowE implements Comparable {
  int [] row; // starts as null

  static int sortColumn = 0;
  static boolean orderAZ = true;
  static int printWidth = 5;
  static Random rn = new Random ();

  public RowE (int c, int m) {
    row = new int [c];
    for (int i = 0; i < c; i++)
      row[i] = rn.nextInt (m);
  } // end random constructor

  public String toString () {
    String st = "";
    for (int x : row)
      st += String.format ("%" + printWidth + "d", x);
    return st;
  } // end toString

  public static void setCol (int     v) {sortColumn = v;}
  public static void setUp  (boolean t) {orderAZ = t;}

  // instance method required by Comparable
  public int compareTo (Object o) {
    if (! (o instanceof RowE)) return 0;
    RowE r = (RowE) o;
    if (orderAZ) return row[sortColumn] - r.row[sortColumn];
    return r.row[sortColumn] - row[sortColumn];
  } // end method compareTo
} // end RowE
```

Arrays.sort (rr) – uses the NATURAL order on the class RowE, which is determined by the compareTo method in RowE.



Sorting E W RowE – 2D int

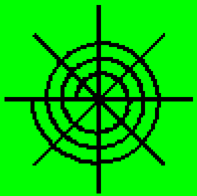
```java
// new class Rows to demonstrate Comparable
// interface made effective use of Arrays.sort
// using Comparable
// NOT thread safe, or multiple instance safe

import java.util.Random;
import java.util.Arrays;

public class MySortE {
  public static void main (String args []) {
    RowE [] rr = newArray (15, 5, 100);
    printArray (rr);

    RowE.setCol (3);      // sort by column 3
    RowE.setUp  (false); // sort descending
    Arrays.sort (rr);

    System.out.println ("--- Sorted:");
    printArray (rr);
  } // end main

  static RowE [] newArray (int r, int c, int m) {
    RowE [] arr = new RowE [r];
    // entries null at this point
    for (int i = 0; i < r; i++)
      arr [i] = new RowE (c, m);
    return arr;
  } // end newArray

  static void printArray (RowE [] a) {
    for (RowE r : a)
      System.out.println (r);
  } // end printArray
} // end class MySortE
```

```java
class RowE implements Comparable {
  int [] row; // starts as null

  static int sortColumn = 0;
  static boolean orderAZ = true;
  static int printWidth = 5;
  static Random rn = new Random ();

  public RowE (int c, int m) {
    row = new int [c];
    for (int i = 0; i < c; i++)
      row[i] = rn.nextInt (m);
  } // end random constructor

  public String toString () {
    String st = "";
    for (int x : row)
      st += String.format ("%" + printWidth + "d", x);
    return st;
  } // end toString

  public static void setCol (int     v) {sortColumn = v;}
  public static void setUp  (boolean t) {orderAZ = t;}

 // instance method required by Comparable
  public int compareTo (Object o) {
    if (! (o instanceof RowE)) return 0;
    RowE r = (RowE) o;
    if (orderAZ) return row[sortColumn] - r.row[sortColumn];
    return r.row[sortColumn] - row[sortColumn];
  } // end method compareTo
} // end RowE
```

newArray is a FACTORY METHOD – meaning that it is not a constructor, but returns a new instance of a class (or in this case, an array of a class, namely an array of RowE).

It starts by creating a array of r elements. At this point, each element is null.

Then each element is initialized to the an instance of RowE. This is done through the (int, int) constructor of the RowE class. We'll look at the constructor on the next slide.



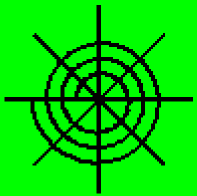
Sorting E W RowE – 2D int

```
// new class Rows to demonstrate Comparable
// interface made effective use of Arrays.sort
// using Comparable
// NOT thread safe, or multiple instance safe

import java.util.Random;
import java.util.Arrays;

public class MySortE {
  public static void main (String args []) {
    RowE [] rr = newArray (15, 5, 100);
    printArray (rr);

    RowE.setCol  (3);      // sort by column 3
    RowE.setUp   (false); // sort descending
    Arrays.sort (rr);

    System.out.println ("--- Sorted:");
    printArray (rr);
  } // end main

  static RowE [] newArray (int r, int c, int m) {
    RowE [] arr = new RowE [r];
    // entries null at this point
    for (int i = 0; i < r; i++)
      arr [i] = new RowE (c, m);
    return arr;
  } // end newArray

  static void printArray (RowE [] a) {
    for (RowE r : a)
      System.out.println (r);
  } // end printArray
} // end class MySortE
```

```
class RowE implements Comparable {
  int [] row; // starts as null

  static int sortColumn = 0;
  static boolean orderAZ = true;
  static int printWidth = 5;
  static Random rn = new Random ();

  public RowE (int c, int m) {
    row = new int [c];
    for (int i = 0; i < c; i++)
      row[i] = rn.nextInt (m);
  } // end random constructor

  public String toString () {
    String st = "";
    for (int x : row)
      st += String.format ("%" + printWidth + "d", x);
    return st;
  } // end toString

  public static void setCol (int     v) {sortColumn = v;}
  public static void setUp  (boolean t) {orderAZ = t;}

  // instance method required by Comparable
  public int compareTo (Object o) {
    if (! (o instanceof RowE)) return 0;
    RowE r = (RowE) o;
    if (orderAZ) return row[sortColumn] - r.row[sortColumn];
    return r.row[sortColumn] - row[sortColumn];
  } // end method compareTo
} // end RowE
```

RowE has 4 methods:

- compareTo – the crux of the sorting system.
- setUp – selects ascending or descending order
- setCol – selects the column for sorting, 0 is the first one
- toString – prints the row of numbers nicely, using format
- RowE (int, int) – a constructor that will create a row of a given size, with random numbers



Sorting E W RowE – 2D int

```
// new class Rows to demonstrate Comparable
// interface made effective use of Arrays.sort
// using Comparable
// NOT thread safe, or multiple instance safe

import java.util.Random;
import java.util.Arrays;

public class MySortE {
  public static void main (String args []) {
    RowE [] rr = newArray (15, 5, 100);
    printArray (rr);

    RowE.setCol (3);      // sort by column 3
    RowE.setUp  (false); // sort descending
    Arrays.sort (rr);

    System.out.println ("--- Sorted:");
    printArray (rr);
  } // end main

  static RowE [] newArray (int r, int c, int m) {
    RowE [] arr = new RowE [r];
    // entries null at this point
    for (int i = 0; i < r; i++)
      arr [i] = new RowE (c, m);
    return arr;
  } // end newArray

  static void printArray (RowE [] a) {
    for (RowE r : a)
      System.out.println (r);
  } // end printArray
} // end class MySortE
```

```
class RowE implements Comparable {
  int [] row; // starts as null

  static int sortColumn = 0;
  static boolean orderAZ = true;
  static int printWidth = 5;
  static Random rn = new Random ();

  public RowE (int c, int m) {
    row = new int [c];
    for (int i = 0; i < c; i++)
      row[i] = rn.nextInt (m);
  } // end random constructor

  public String toString () {
    String st = "";
    for (int x : row)
      st += String.format ("%" + printWidth + "d", x);
    return st;
  } // end toString

  public static void setCol (int     v) {sortColumn = v;}
  public static void setUp  (boolean t) {orderAZ = t;}

 // instance method required by Comparable
  public int compareTo (Object o) {
    if (! (o instanceof RowE)) return 0;
    RowE r = (RowE) o;
    if (orderAZ) return row[sortColumn] - r.row[sortColumn];
    return r.row[sortColumn] - row[sortColumn];
  } // end method compareTo
} // end RowE
```

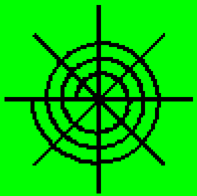
compareTo – this method is used by Arrays.sort and other sorting algorithms to compare two elements and put them in the proper order.

Without generics, the parameter type is Object, but really only RowE is OK. Thus, this method really should throw an exception if the parameter is the wrong type.

The last two lines do the real work, based on the order.

Note that this is based on the difference between:

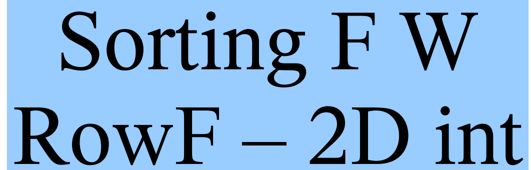
this.row – r.row (instance – parameter)
r.row – this.row (parameter – instance)



Sorting F W RowF – 2D int

```java
// new class Rows to demonstrate Comparator interface
// made effective use of Arrays.sort using Comparators
// IS thread safe, and multiple instance safe

import java.util.Random;
import java.util.Arrays;
import java.util.Comparator;

public class MySortF {
  public static void main (String args []) {
    RowF [] rr = newArray (15, 5, 1000);
    printArray (rr);

    System.out.println ("--- Sorted by column 0:");
    Arrays.sort (rr, RowF.COL0); printArray (rr);

    System.out.println ("--- Sorted by column 1:");
    Arrays.sort (rr, RowF.COL1); printArray (rr);

    System.out.println ("--- Sorted by column 2:");
    Arrays.sort (rr, RowF.COL2); printArray (rr);

    System.out.println ("--- Sorted by column 3:");
    Arrays.sort (rr, RowF.COL3); printArray (rr);

  } // end main

  static RowF [] newArray (int r, int c, int m) {
    RowF [] arr = new RowF [r];
      // entries null at this point
    for (int i = 0; i < r; i++)
      arr [i] = new RowF (c, m);
    return arr;
  } // end newArray

  static void printArray (RowF [] a) {
    for (RowF r : a)
      System.out.println (r);
  } // end printArray
} // end class MySortE
```

```java
class RowF {
  int [] row; // starts as null

  static int printWidth = 5;
  static Random rn = new Random ();

  public RowF (int c, int m) {
    row = new int [c];
    for (int i = 0; i < c; i++)
      row[i] = rn.nextInt (m);
  } // end random constructor

  public String toString () {
    String st = "";
    for (int x : row)
      st += String.format ("%" + printWidth + "d", x);
    return st;
  } // end toString

  // using anonymous inner class without generics
  // casting required, and Xlint complains
  static Comparator COL0 = new Comparator () {
    public int compare (Object o1, Object o2) {
      return ((RowF)o1).row[0] - ((RowF)o2).row[0];
    } // end interface method compare
  }; // semi-colon required

  // using anonymous inner class and generics
  static Comparator <RowF> COL1 = new Comparator <RowF> () {
    public int compare (RowF r1, RowF r2) {
      return r1.row[1] - r2.row[1];
    } // end interface method compare
  }; // end inline definition of class

  // using a named static inner class definition, no generics
  // casting required, and Xlint complains
  static Col2 COL2 = new Col2 ();
  static class Col2 implements Comparator {
    public int compare (Object o1, Object o2) {
      return ((RowF)o1).row[2] - ((RowF)o2).row[2];
    } // end interface method compare
  } // end inner static class

  // using a named static inner class definition, with generics
  static Col3 COL3 = new Col3 ();
  static class Col3 implements Comparator <RowF> {
    public int compare (RowF r1, RowF r2) {
      return r1.row[3] - r2.row[3];
    } // end interface method compare
  } // end inner static class

} // end class RowF
```



Sorting F W RowF – 2D int

```java
// new class Rows to demonstrate Comparator interface
// made effective use of Arrays.sort using Comparators
// IS thread safe, and multiple instance safe

import java.util.Random;
import java.util.Arrays;
import java.util.Comparator;

public class MySortF {
  public static void main (String args []) {
    RowF [] rr = newArray (15, 5, 1000);
    printArray (rr);

    System.out.println ("--- Sorted by column 0:");
    Arrays.sort (rr, RowF.COL0); printArray (rr);

    System.out.println ("--- Sorted by column 1:");
    Arrays.sort (rr, RowF.COL1); printArray (rr);

    System.out.println ("--- Sorted by column 2:");
    Arrays.sort (rr, RowF.COL2); printArray (rr);

    System.out.println ("--- Sorted by column 3:");
    Arrays.sort (rr, RowF.COL3); printArray (rr);

  } // end main

  static RowF [] newArray (int r, int c, int m) {
    RowF [] arr = new RowF [r];
      // entries null at this point
    for (int i = 0; i < r; i++)
      arr [i] = new RowF (c, m);
    return arr;
  } // end newArray

  static void printArray (RowF [] a) {
    for (RowF r : a)
      System.out.println (r);
  } // end printArray
} // end class MySortE
```

This is a more complicated example in which one class, RowF, has 4 comparators associated with it, and the Arrays.sort method now has two parameters: the array to be sorted (as before) and the comparator that is to be used in this particular sort.

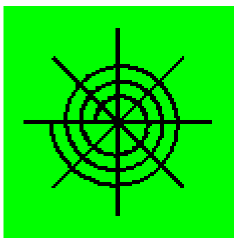
The class RowF has 4 static variables (COL0, COL1, COL2, and COL3), which are instances of classes, each of which implements the Comparator interface.

On the next slides, we will explore how these instances are defined and how they work.

main in MySortF is pretty much the same as we have seen before:

- Create and init a 2-D array
- Print the array
- Sort the array and print it – now 4 different times, using 4 different sorts – 4 different columns

The other two methods (newArray and printArray) are the same as before.



Sorting F RowF – 2D int

The class RowF:

Note that the main class here, RowF, DOES NOT implement Comparator – the inner classes are going to do that.

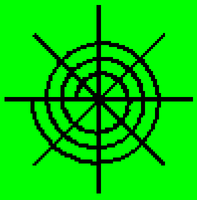
This class has the same instance variable as before – a 1-D array of ints. The approach we use here does not use static variables to select the column or order, but they, or similar variables, could be used here also.

The constructor and the toString method are the same as before.

This program shows 4 different approaches to defining comparators for a class.

- Anonymous inner class
- Anonymous inner class with generics
- Named inner class
- Named inner class with generics

```
class RowF {
  int [] row; // starts as null
  static int printWidth = 5;
  static Random rn = new Random ();

  public RowF (int c, int m) {
    row = new int [c];
    for (int i = 0; i < c; i++)
      row[i] = rn.nextInt (m);
  } // end random constructor

  public String toString () {
    String st = "";
    for (int x : row)
      st += String.format ("%" + printWidth + "d", x);
    return st;
  } // end toString

  // using anonymous inner class without generics
  // casting required, and Xlint complains
  static Comparator COL0 = new Comparator () {
    public int compare (Object o1, Object o2) {
      return ((RowF)o1).row[0] - ((RowF)o2).row[0];
    } // end interface method compare
  }; // semi-colon required

  // using anonymous inner class and generics
  static Comparator <RowF> COL1 = new Comparator <RowF> () {
    public int compare (RowF r1, RowF r2) {
      return r1.row[1] - r2.row[1];
    } // end interface method compare
  }; // end inline definition of class

  // using a named static inner class definition, no generics
  // casting required, and Xlint complains
  static Col2 COL2 = new Col2 ();
  static class Col2 implements Comparator {
    public int compare (Object o1, Object o2) {
      return ((RowF)o1).row[2] - ((RowF)o2).row[2];
    } // end interface method compare
  } // end inner static class

  // using a named static inner class definition, with generics
  static Col3 COL3 = new Col3 ();
  static class Col3 implements Comparator <RowF> {
    public int compare (RowF r1, RowF r2) {
      return r1.row[3] - r2.row[3];
    } // end interface method compare
  } // end inner static class

} // end class RowF
```



Sorting F RowF – Col2

Col2 – we'll start with the simplest Comparator first.

This line declares a static variable, COL2, of type Col2, and gives it a value. The value of the variable is a new instance of Col2, which will be defined in the next few lines. Nothing really special about this line.

"static" – Because the two instances of RowF to be compared are passed as parameters, there is no need for an instance context ("this"), thus the variable "COL2" should be static, and if the class definition is inside the class RowF, it must also be static. The definition of Col2 could be outside RowF, in which case, the class definition is not allowed to be static.

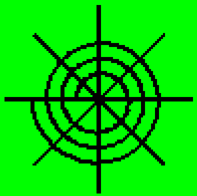
"implements Comparator"- well, that's the point, isn't it?

"Object" – annoying, but this is required without generics.

Casting: also required. Notice the parentheses.

The ordering is by the third elements in the arrays (row[2]).

```
// using a named static inner class definition, no generics
// casting required, and Xlint complains

static Col2 COL2 = new Col2 ();

static class Col2 implements Comparator {

  public int compare (Object o1, Object o2) {

    return ((RowF)o1).row[2] - ((RowF)o2).row[2];

  } // end interface method compare

} // end inner static class
```



Sorting F
RowF – Col3

Col3 is very much like Col2, except that by taking advantage of generics we eliminate the need for casting, and avoid the nasty "uses unchecked or unsafe operations" compiler warning.

Here is where the generic declaration goes.

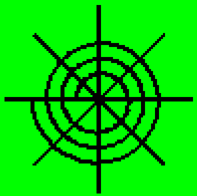
Now, we can use RowF as the parameter type, rather than Object.

We don't need to cast.

The reason this works is that the method
Arrays.sort (T e[], Comparator <T> c)
is also parameterized using generics. In this notation, the generic parameter is T, the class that will be used in the sort method.

Note that the SAME T is used in both parameters. This allows the compiler to check that the array of T-type instances is to be compared using a Comparator of T-type instances.

```
// using a named static inner class definition, with generics

  static Col3 COL3 = new Col3 ();

  static class Col3 implements Comparator <RowF> {

    public int compare (RowF r1, RowF r2) {

      return r1.row[3] - r2.row[3];

    } // end interface method compare
  } // end inner static class
```



Sorting F RowF – Col0

Col0 – using an anonymous inner class.

The first part declares a new static variable, COL0, of the interface type Comparator.

The second part, starting with "new", instantiates a anonymous inner class that implements the current interface Comparator.

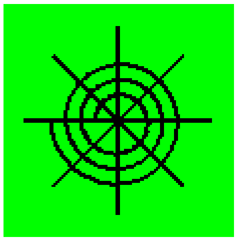
The definition of this class is between the curly braces. Note how the syntax works – up to the start of the braces, this looks like the code needed to instantiate (new) any regular class.

"compare" is, of course, the method that the interface Comparator requires.

As with Col2, since this is not a generic definition, we need to use Object as the parameter type, and explicit casting in the code.

The business part of the code, the "return", is the same as Col2, except that it sorts on the first column (row[0]).

```
// using anonymous inner class without generics
// casting required, and Xlint complains

static Comparator COL0 = new Comparator ()      {

public int compare (Object o1, Object o2) {

  return ((RowF)o1).row[0] - ((RowF)o2).row[0];

 } // end interface method compare

}       ; // semi-colon required
```



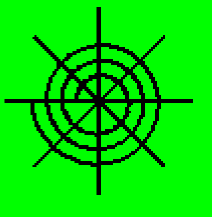
Sorting F RowF – Col1

Col1 is pretty much the same as Col0, but we have added the generic declaration in two places.

We can now use the actual type, RowF, in the parameter list.

We no longer need casting the internal computations.

```
// using anonymous inner class and generics
static Comparator <RowF> COL1 =

      new Comparator <RowF> ()
{
  public int compare (RowF r1, RowF r2) {

      return r1.row[1] - r2.row[1];

  } // end interface method compare

}     ; // end inline definition of class
```



Sorting G Desc. String

MySortG is pretty much the same as MySortB – sorts an array of String's, in this case, 50 random 4-character strings.

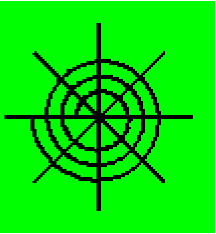
Two differences:

(a) The call to Array.sort has a new parameter, SORTG, as we have already seen in MySortF.

(b) There is a new class, SortG, which defines a different order on the class String (reverse order).

SortG – this code uses the named generic inner class, parameterized with the type String. Because of the parameterization, we can use String in the "compare" method, and we don't need to cast in the method itself.

```
import java.util.Random;
import java.util.Arrays;
import java.util.Comparator;

public class MySortG {
   static Random rn = new Random ();

   public static void printArray (String a [], int cols) {
      int i = 0;
      for (String st: a) {
         if (i++ % cols == 0) System.out.println ();
         System.out.printf ("%" + (st.length() + 1) + "s", st);
      } // end for each element in array
   } // end printArray

  public static String [] randomStrings (int num, int len) {
      String x [] = new String [num];
      char c [] = new char [len];

      for (int i = 0; i < x.length; i++) {
         for (int j = 0; j < c.length; j++)
            c[j] = (char) ('a' + rn.nextInt (26));
         x[i] = new String (c);
      } // end for each array element

      return x;
   } // end method randomStrings

   public static void main (String args []) {
      String x [] = randomStrings (50, 4);
      printArray (x, 10);
      Arrays.sort (x, SORTG); // x must implement Comparable
      System.out.println ("\n\nSorted, reverse order:");
      printArray (x, 10);
   } // end main

     // using a named static inner class definition, with generics
   static SortG SORTG = new SortG ();
   static class SortG implements Comparator <String> {
      public int compare (String r1, String r2) {
         return r2.compareTo(r1);
      } // end interface method compare
   } // end inner static class

} // end class MySortG
```



Sorting G Desc. String

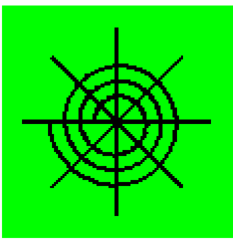
SORTG – the static instance of the class SortG used as the parameter to Arrays.sort (T e[], Comparator <T> c). In this case, the generic parameter T is in fact of type String – and the compiler WILL check this.

Here the generic parameter to the Comparator class set to String in the SortG declaration.

And because of the generic declaration of the class SortG, the compare method can use the String class directly, no need for any casting.

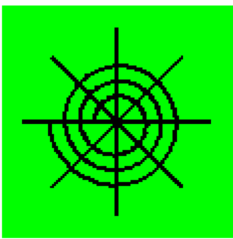
Finally, the actual calculation uses the String.compareTo (String) method of the String class, but in reverse order!

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

    Arrays.sort (x, SORTG); // x must implement Comparable

  } // end main


    // using a named static inner class definition, with generics
  static SortG SORTG = new SortG ();


  static class SortG implements Comparator <String> {


    public int compare (String r1, String r2) {
      return r2.compareTo(r1);
    } // end interface method compare
  } // end inner static class

} // end class MySortG
```



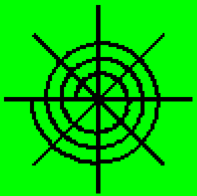
Vector

- Vector is a very powerful class introduced in JDK 1.0, but it has its limitations. We begin by exploring the key features and limitations of the class and then we will consider a few examples.
- From the API: Vector implements a growable (and shrinkable) array of objects. Key points to emphasize:
  - The size of this structure is automatically adjusted for the programmer.
  - The elements are type Object.
    - Primitive data types, such as int and double, may not be elements.
    - Insert operations (add) are trivial, since anything other than a primitive data type is an Object.
    - Get operations only return a type Object, which can cause compiler errors. Those errors can only be addressed by explicit type casting – circumventing type checking.



Vector - Versions

- As we noted on the previous slide, Vector has been around since JDK 1.0, but with the advent of JDK 5.0 and generic (parameterized) classes, the class has moved from a class whose parent was Object to a class which extends AbstractList, which in turns extends AbstractCollection.
- For the first set of examples, we will use the original version of Vector. The purpose of these examples is to explore the typecasting issues, and give some of the motivation for generics.
- The Vector class now includes
  - 4 protected fields
  - 4 constructors
  - 42 methods in this class
  - 14 inherited methods from Object, java.util.AbstractList and java.util.List
- We will only have a chance to examine some of the basic methods.

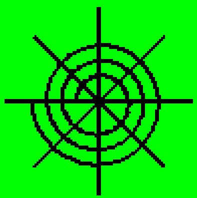


Vector Mixed -A

- Vector – primary methods
  - add
  - get
  - indexOf
  - remove, clear
  - size
  - toString

```
import java.util.Vector;
import java.awt.Point;

class MyVectorDemoA {
     public static void main (String args []) {
          Vector v = new Vector ();
          v.add (new String ("this is the first"));
          v.add (new Point (12, 33));
          v.add (new String ("the third entry"));
          System.out.println ("The vector:\n" + v);
     } // end main
} // end class MyVectorDemoA

// Output:
/*
 The vector:
[this is the first, java.awt.Point[x=12,y=33], the third entry]
*/
```

This is a very simple example that demonstrates the flexibility of the Vector class.
Notice that this example puts objects of type String and Point into the same Vector.
The 5.0 compiler gives warnings about unchecked calls to “add”, but this code works nicely.
Notice also the implicit use of toString in println, which then calls toString on each element in the Vector.



Vector Error! - B

- An error in referencing data from a Vector
- This code compiles, but this line gives a runtime error, since v[1] is actually a Point.
- This line of code would be OK, since after the toString, the RHS is in fact a String.

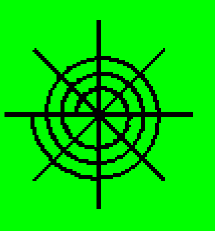
```
// File: MyVectorDemoB

import java.util.Vector;
import java.awt.Point;

class MyVectorDemoB {
   public static void main (String args []) {
      Vector v = new Vector ();
      v.add (new String ("One"));
      v.add (new Point (12, 33));
      v.add (new String ("two"));
      System.out.println ("Vector:\n" + v + "\n");
      String a = (String) v.get(0);

      String b = (String) v.get(1);

//       String b = (String) v.get(1).toString();

      String c = (String) v.get(2);
      System.out.printf ("a, b, c: %s, %s, %s\n", a, b, c);
   } // end main
} // end class MyVectorDemoB
```

This is a typical example where the compiler accepts the promise made by the programmer, but the RTE will check the data types at run time and flag errors as it sees them.

This error is precisely the problem the compiler is warning about when it talks about unchecked calls.

Question A: In the corrected code (activate commented line and deactivate the line that gives the run-time error), what is the value of "b"? In other words, explain the output of this program.

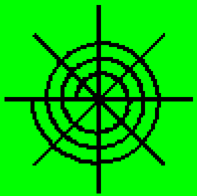
Question B: Using the following compiler line, explain the 3 warnings this code generates:
> javac -Xlint MyVectorDemoB.java



Collections Basic – C W

- Working

```
// File: MyVectorDemoC
// Vector of String

   import java.util.Vector;
   import java.util.ArrayList;
   import java.util.LinkedList;
   import java.util.Collections;
   import java.util.List;
//    import java.awt.Point;

   class MyListC {
      public static void main (String args []) {
//       Vector     <String> v = new Vector     <String> ();
//       ArrayList  <String> v = new ArrayList  <String> ();
         LinkedList <String> v = new LinkedList <String> ();
         v.add ("abcd");
         // the following is a compiler error, from generic
//          v.add (new Point (12, 33));
         v.add ("mnop");
         v.add ("efgh");
         System.out.println ("The vector:\n" + v);
         Collections.sort (v);
         System.out.println ("The sorted vector:\n" + v);
      } // end main
   } // end class MyVectorDemoA

   // To make generics work correctly, v must be declared
   // generic, and it is the COMPILER that enforces the
   // generic type checking, NOT the RTE.
```

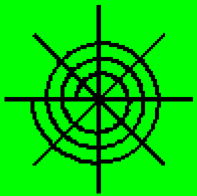


Collections Strings - D

```
// File: MyListD
// Using built-in data structures
// Structures of Strings, reversed

import java.util.List;         // Interface

import java.util.Vector;       // concrete class
import java.util.ArrayList;    // concrete class
import java.util.LinkedList;   // concrete class

import java.util.Comparator;   // interface

import java.util.Collections;  // concrete class

import java.util.Random;       // concrete class

public class MyListD {
   static Random rn = new Random ();

   public static void printArray
         (Iterable <String> a, int cols) {
      int i = 0;
      for (String st: a) {
         if (i++ % cols == 0) System.out.println ();
         System.out.printf ("%"+(st.length()+1)+"s", st);
      } // end for each element in array
   } // end printArray

   public static List <String> randomStrings
         (int num, int len) {
//    List <String> x = new Vector     <String> ();
//    List <String> x = new ArrayList  <String> ();
      List <String> x = new LinkedList <String> ();
      char c [] = new char [len];

      for (int i = 0; i < num; i++) {
         for (int j = 0; j < c.length; j++)
            c[j] = (char) ('a' + rn.nextInt (26));
         x.add(new String (c));
      } // end for each array element

      return x;
   } // end method randomStrings

   public static void main (String args []) {
      // AbstractList ok, but NOT AbstractCollection
      List <String> x = randomStrings (50, 4);

      printArray (x, 10);
      // x must implement Comparable
      Collections.sort (x, SORTG);
      System.out.println ("\n\nSorted, reverse order:");
      printArray (x, 10);

      Collections.shuffle (x);
      System.out.println ("\n\nShuffled:");
      printArray (x, 10);

      Collections.sort (x);
      System.out.println ("\n\nSorted:");
      printArray (x, 10);
   } // end main

   // using a named static inner class definition
   // with generics
   static SortG SORTG = new SortG ();
   static class SortG implements Comparator <String> {
      public int compare (String r1, String r2) {
         return r2.compareTo(r1);
      } // end interface method compare
   } // end inner static class

} // end class MySortG
```

MyListD is based on the code in MySortG, with some modifications that let this class use a number of more interesting data structures in the Collections classes: Vector, ArrayList and LinkedList.

This example also demonstrates the difference between the using the natural ordering and a custom ordering in a sort.



Collections Strings - D

```
// File: MyListD
// Using built-in data structures
// Structures of Strings, reversed

import java.util.List;          // Interface

import java.util.Vector;        // concrete class
import java.util.ArrayList;     // concrete class
import java.util.LinkedList;    // concrete class

import java.util.Comparator;    // interface

import java.util.Collections;   // concrete class

import java.util.Random;        // concrete class

public class MyListD {
   static Random rn = new Random ();

   public static void printArray
         (Iterable <String> a, int cols) {
      int i = 0;
      for (String st: a) {
         if (i++ % cols == 0) System.out.println ();
         System.out.printf ("%"+(st.length()+1)+"s", st);
      } // end for each element in array
   } // end printArray

   public static List <String> randomStrings
         (int num, int len) {
//    List <String> x = new Vector     <String> ();
//    List <String> x = new ArrayList  <String> ();
      List <String> x = new LinkedList <String> ();
      char c [] = new char [len];

      for (int i = 0; i < num; i++) {
         for (int j = 0; j < c.length; j++)
            c[j] = (char) ('a' + rn.nextInt (26));
         x.add(new String (c));
      } // end for each array element

      return x;
   } // end method randomStrings
```

```
   public static void main (String args []) {
      // AbstractList ok, but NOT AbstractCollection
      List <String> x = randomStrings (50, 4);
// Stuff
   } // end main
// More Stuff
} // end class MySortG
```

Interfaces – let's start with a discussion of interfaces. In this program, we have used the most general declarations for the variable types (local and parameter) that work:

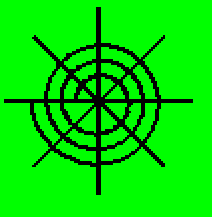
- java.util.List
- java.lang.Iterable

Since the Iterable interface is part of java.lang, it is automatically imported and does not need and explicit import statement, and it is required for the advanced for loop in printArray.

Recall the following:

List extends Collection extends Iterable

And that the three concrete methods used in this example, Vector, ArrayList and LinkedList, all implement the List interface. This is and issue because the Collections.sort method requires the first parameter to implement the List interface (ie, it must be type List).

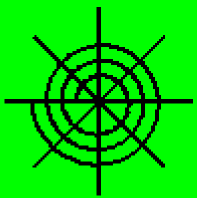


Collections Strings - D

```
// File: MyListD
// Using built-in data structures
// Structures of Strings, reversed

import java.util.List;          // Interface

import java.util.Vector;        // concrete class
import java.util.ArrayList;     // concrete class
import java.util.LinkedList;    // concrete class

import java.util.Comparator;    // interface

import java.util.Collections;   // concrete class

import java.util.Random;        // concrete class

public class MyListD {
   static Random rn = new Random ();

   public static void printArray
         (Iterable <String> a, int cols) {
      int i = 0;
      for (String st: a) {
         if (i++ % cols == 0) System.out.println ();
         System.out.printf ("%"+(st.length()+1)+"s", st);
      } // end for each element in array
   } // end printArray

   public static List <String> randomStrings
         (int num, int len) {
//    List <String> x = new Vector     <String> ();
//    List <String> x = new ArrayList  <String> ();
      List <String> x = new LinkedList <String> ();
      char c [] = new char [len];

      for (int i = 0; i < num; i++) {
         for (int j = 0; j < c.length; j++)
            c[j] = (char) ('a' + rn.nextInt (26));
         x.add(new String (c));
      } // end for each array element

      return x;
   } // end method randomStrings

   public static void main (String args []) {
      // AbstractList ok, but NOT AbstractCollection
      List <String> x = randomStrings (50, 4);

      printArray (x, 10);
      // x must implement Comparable
      Collections.sort (x, SORTG);
      System.out.println ("\n\nSorted, reverse order:");
      printArray (x, 10);

      Collections.shuffle (x);
      System.out.println ("\n\nShuffled:");
      printArray (x, 10);

      Collections.sort (x);
      System.out.println ("\n\nSorted:");
      printArray (x, 10);
   } // end main

   // using a named static inner class definition
   // with generics
   static SortG SORTG = new SortG ();
   static class SortG implements Comparator <String> {
      public int compare (String r1, String r2) {
         return r2.compareTo(r1);
      } // end interface method compare
   } // end inner static class

} // end class MySortG
```

Interfaces are used heavily in this example, so let us mark them on this slide.

On the next slides, we will show how the declarations fit together with how they are used, and we will delete the imports section to get a little more space for comments.



Collections Strings - D

Let's start with the easy one: Iterable <String> in printArray.

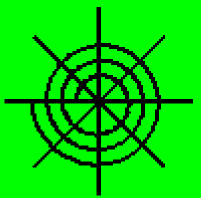
The advanced for loop requires, as a minimum, that the data structure to be iterated over implements Iterable. The generic declaration String allows the code to declare the variable in the for loop to be type String, and that allows the specification st.length ().

```
public class MyListD {
   static Random rn = new Random ();

   public static void printArray
         (Iterable <String> a, int cols) {
      int i = 0;
      for (String st: a) {
         if (i++ % cols == 0) System.out.println ();
         System.out.printf ("%"+(st.length()+1)+"s", st);
      } // end for each element in array
   } // end printArray

   public static List <String> randomStrings
         (int num, int len) {
//    List <String> x = new Vector     <String> ();
//    List <String> x = new ArrayList  <String> ();
      List <String> x = new LinkedList <String> ();
      char c [] = new char [len];

      for (int i = 0; i < num; i++) {
         for (int j = 0; j < c.length; j++)
            c[j] = (char) ('a' + rn.nextInt (26));
         x.add(new String (c));
      } // end for each array element

      return x;
   } // end method randomStrings
```

In main, the call to printArray needs an instance of a class that implements Iterable, and since the class List <String> extends Iterable <String>, the compiler can conclude that "x" in main has the right type.

The point of using Iterable is that the code does NOT need to specify exactly what data structure is being used.

```
   public static void main (String args []) {
      // AbstractList ok, but NOT AbstractCollection
      List <String> x = randomStrings (50, 4);

      printArray (x, 10);
      // x must implement Comparable
      Collections.sort (x, SORTG);
      System.out.println ("\n\nSorted, reverse order:");
      printArray (x, 10);

      Collections.shuffle (x);
      System.out.println ("\n\nShuffled:");
      printArray (x, 10);

      Collections.sort (x);
      System.out.println ("\n\nSorted:");
      printArray (x, 10);
   } // end main

   // using a named static inner class definition
   // with generics
   static SortG SORTG = new SortG ();
   static class SortG implements Comparator <String> {
      public int compare (String r1, String r2) {
         return r2.compareTo(r1);
      } // end interface method compare
   } // end inner static class

} // end class MySortG
```



Collections Strings - D

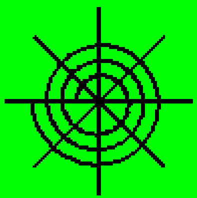
Since the local variable "x" in main is declared type List <String>, the method randomStrings must return an object that implements List, which gives us the reason for the return type of the randomStrings method, as well as the the type for local variable "x" in that method.

```
public class MyListD {
   static Random rn = new Random ();

   public static void printArray
         (Iterable <String> a, int cols) {
      int i = 0;
      for (String st: a) {
         if (i++ % cols == 0) System.out.println ();
         System.out.printf ("%"+(st.length()+1)+"s", st);
      } // end for each element in array
   } // end printArray

   public static List <String> randomStrings
         (int num, int len) {
//    List <String> x = new Vector     <String> ();
//    List <String> x = new ArrayList  <String> ();
      List <String> x = new LinkedList <String> ();
      char c [] = new char [len];

      for (int i = 0; i < num; i++) {
         for (int j = 0; j < c.length; j++)
            c[j] = (char) ('a' + rn.nextInt (26));
         x.add(new String (c));
      } // end for each array element

      return x;
   } // end method randomStrings
```

Collections.sort and Collections.shuffle require a structure that implements the List interface, thus, x is declared as a generic (<String>) List.

This means that randomStrings must return an object that implements List, which gives us the reason for the return type of the randomStrings method, as well as the the type for local variable "x" in that method.

```
   public static void main (String args []) {
      // AbstractList ok, but NOT AbstractCollection
      List <String> x = randomStrings (50, 4);

      printArray (x, 10);
      // x must implement Comparable
      Collections.sort (x, SORTG);
      System.out.println ("\n\nSorted, reverse order:");
      printArray (x, 10);

      Collections.shuffle (x);
      System.out.println ("\n\nShuffled:");
      printArray (x, 10);

      Collections.sort (x);
      System.out.println ("\n\nSorted:");
      printArray (x, 10);
   } // end main

   // using a named static inner class definition
   // with generics
   static SortG SORTG = new SortG ();
   static class SortG implements Comparator <String> {
      public int compare (String r1, String r2) {
         return r2.compareTo(r1);
      } // end interface method compare
   } // end inner static class

} // end class MySortG
```



Collections Strings - D

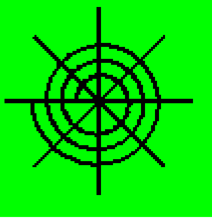
Next, we simply note that the comparisons used to sort this class are just the default (natural order on String), and SortG, the reverse order on the String defined in this example, and which is identical to the SortG class defined in MySortG above.

```
public class MyListD {
   static Random rn = new Random ();

   public static void printArray
         (Iterable <String> a, int cols) {
      int i = 0;
      for (String st: a) {
         if (i++ % cols == 0) System.out.println ();
         System.out.printf ("%"+(st.length()+1)+"s", st);
      } // end for each element in array
   } // end printArray

   public static List <String> randomStrings
         (int num, int len) {
//    List <String> x = new Vector     <String> ();
//    List <String> x = new ArrayList  <String> ();
      List <String> x = new LinkedList <String> ();
      char c [] = new char [len];

      for (int i = 0; i < num; i++) {
         for (int j = 0; j < c.length; j++)
            c[j] = (char) ('a' + rn.nextInt (26));
         x.add(new String (c));
      } // end for each array element

      return x;
   } // end method randomStrings
```

This line uses SortG.

This line uses the default compareTo method in String, giving the natural order of the String class.

```
   public static void main (String args []) {
      // AbstractList ok, but NOT AbstractCollection
      List <String> x = randomStrings (50, 4);

      printArray (x, 10);
      // x must implement Comparable
      Collections.sort (x, SORTG);
      System.out.println ("\n\nSorted, reverse order:");
      printArray (x, 10);

      Collections.shuffle (x);
      System.out.println ("\n\nShuffled:");
      printArray (x, 10);

      Collections.sort (x);
      System.out.println ("\n\nSorted:");
      printArray (x, 10);
   } // end main

   // using a named static inner class definition
   // with generics
   static SortG SORTG = new SortG ();
   static class SortG implements Comparator <String> {
      public int compare (String r1, String r2) {
         return r2.compareTo(r1);
      } // end interface method compare
   } // end inner static class

} // end class MySortG
```



Collections Strings - D

The POINT: Now we get to the point of this entire example: the selection of the concrete data structure used to store the list of String's.

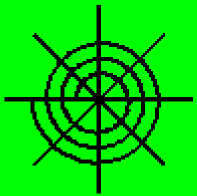
Any concrete class the implements List <String> will do, and we show 3 possibilities:

```
public class MyListD {
   static Random rn = new Random ();

   public static void printArray
         (Iterable <String> a, int cols) {
      int i = 0;
      for (String st: a) {
         if (i++ % cols == 0) System.out.println ();
         System.out.printf ("%"+(st.length()+1)+"s", st);
      } // end for each element in array
   } // end printArray

   public static List <String> randomStrings
         (int num, int len) {
//    List <String> x = new Vector     <String> ();
//    List <String> x = new ArrayList  <String> ();
      List <String> x = new LinkedList <String> ();
      char c [] = new char [len];

      for (int i = 0; i < num; i++) {
         for (int j = 0; j < c.length; j++)
            c[j] = (char) ('a' + rn.nextInt (26));
         x.add(new String (c));
      } // end for each array element

      return x;
   } // end method randomStrings
```

- Vector <String>
- ArrayList <String>
- LinkedList <String>

It is beyond the scope of this lecture to explore these data structures, but we CAN see how easy it is to pick one, and to change one's mind!

```
   public static void main (String args []) {
      // AbstractList ok, but NOT AbstractCollection
      List <String> x = randomStrings (50, 4);

      printArray (x, 10);
      // x must implement Comparable
      Collections.sort (x, SORTG);
      System.out.println ("\n\nSorted, reverse order:");
      printArray (x, 10);

      Collections.shuffle (x);
      System.out.println ("\n\nShuffled:");
      printArray (x, 10);

      Collections.sort (x);
      System.out.println ("\n\nSorted:");
      printArray (x, 10);
   } // end main

   // using a named static inner class definition
   // with generics
   static SortG SORTG = new SortG ();
   static class SortG implements Comparator <String> {
      public int compare (String r1, String r2) {
         return r2.compareTo(r1);
      } // end interface method compare
   } // end inner static class

} // end class MySortG
```



Collections Strings - D

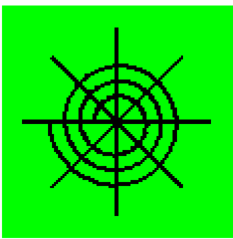
Just to show how flexible this code is, here we have made the selection of which data structure is to be used a run-time decision based on a random number. Each time this code is run, it selects the data structure using a random number generator.

This code will compile without ANY compiler errors or warnings – thus we have given the compiler enough information for it to be quite certain that there will not be any run-time errors due to type mismatches in this code.

And, changing our minds about the actual data structure is now EASY!

Also, note the way we can display the name of the class of an object.

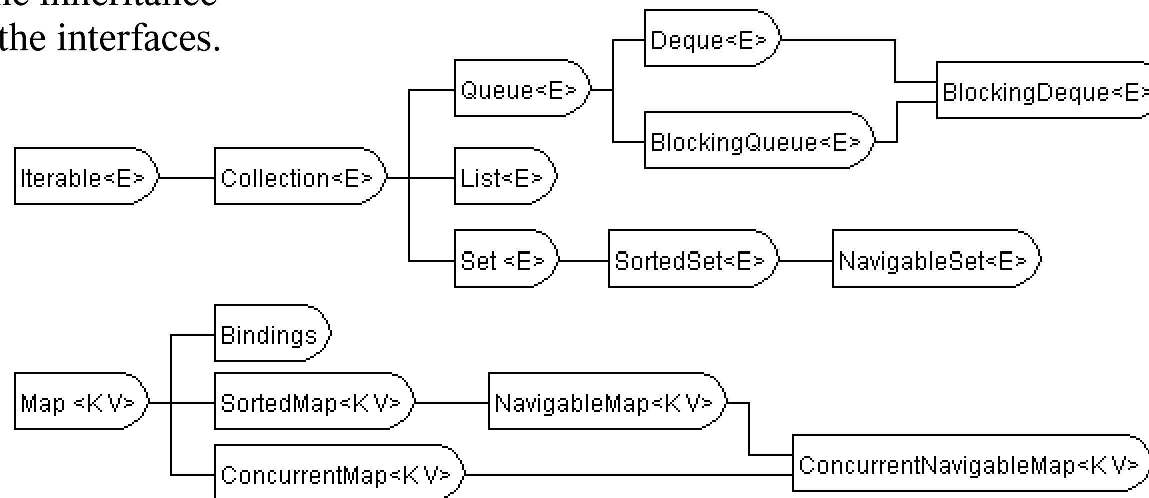
```
public static List <String> randomStrings
      (int num, int len) {

   List <String> x; // null here!
   switch (rn.nextInt (3)) {
     case 0:   x = new Vector     <String> (); break;
     case 1:   x = new ArrayList  <String> (); break;
     default: x = new LinkedList <String> (); break;
   } // end switch

   char c [] = new char [len];

   for (int i = 0; i < num; i++) {
      for (int j = 0; j < c.length; j++)
         c[j] = (char) ('a' + rn.nextInt (26));
      x.add(new String (c));
   } // end for each array element

   return x;
} // end method randomStrings

// in main: displays the class of the data structure:
System.out.println ("\n\nStructure type: " +
     x.getClass().getName ());
```

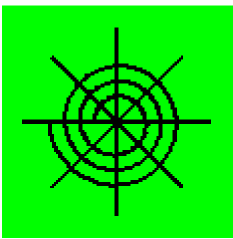


Collections Interfaces
Common Set

These are the general purpose interfaces in JDK that are part of the Collection and Map classes.

This diagram show the inheritance relationships among the interfaces.





Collections Interfaces and Classes

These are the general purpose interfaces and classes in JDK that are part of the Collection class and its children. The abstract and concrete classes implement the interface each is below in this diagram.

Iterable<E>
Collection<E>
AbstractCollection

Queue<E>
AbstractQueue
ConcurrentLinkedQueue
PriorityQueue

Deque<E>
ArrayDeque
LinkedList*

BlockingDeque<E>
LinkedBlockingDeque

BlockingQueue<E>
ArrayBlockingQueue
DelayQueue
LinkedBlockingQueue
PriorityBlockingQueue
SynchronousQueue

List<E>
AbstractList
AbstractSequentialList
ArrayList
AttributeList
CopyOnWriteArrayList
LinkedList*
RoleList
RoleUnresolvedList
Stack
Vector

Set <E>
AbstractSet
CopyOnWriteArraySet
EnumSet
HashSet
JobStateReasons
LinkedHashSet

SortedSet<E>

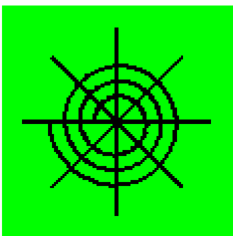
NavigableSet<E>
ConcurrentSkipListSet
TreeSet

Interfaces

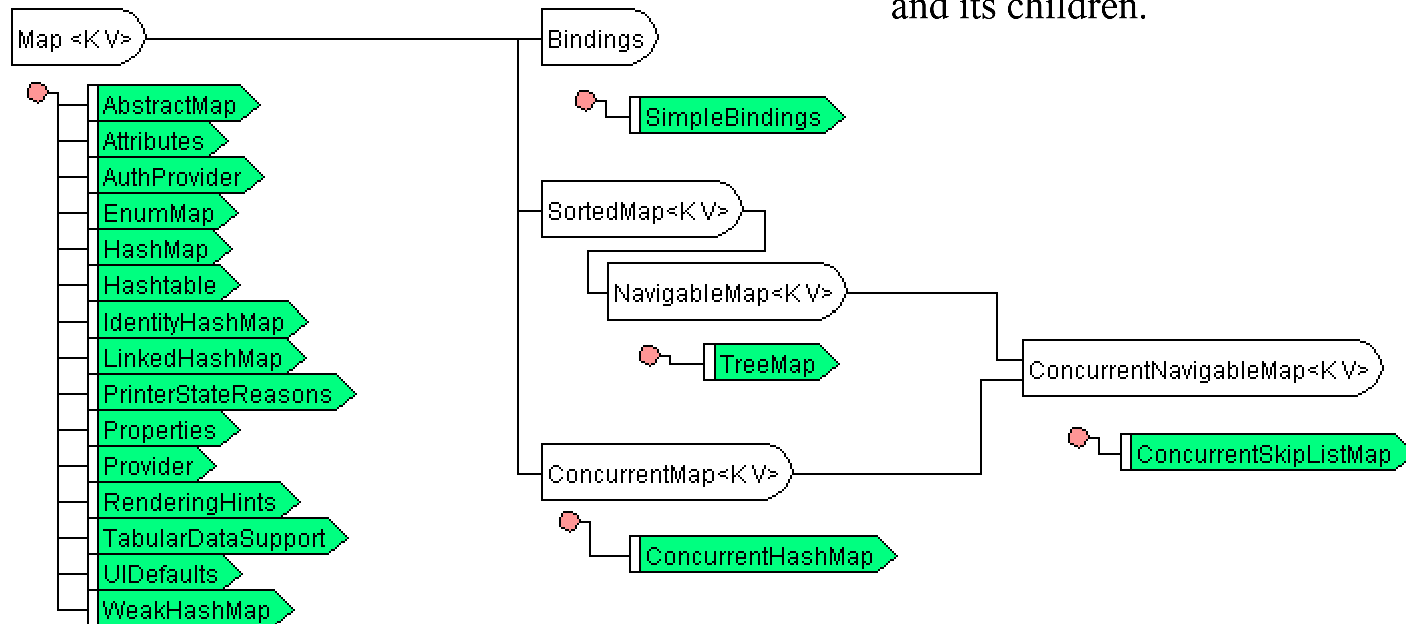
Abstract classes

Concrete classes

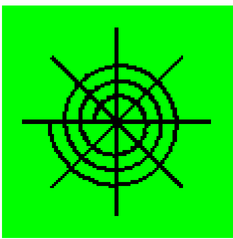
Note that the class LinkedList shows up twice on this diagram, since it implements both the Deque and List interfaces.



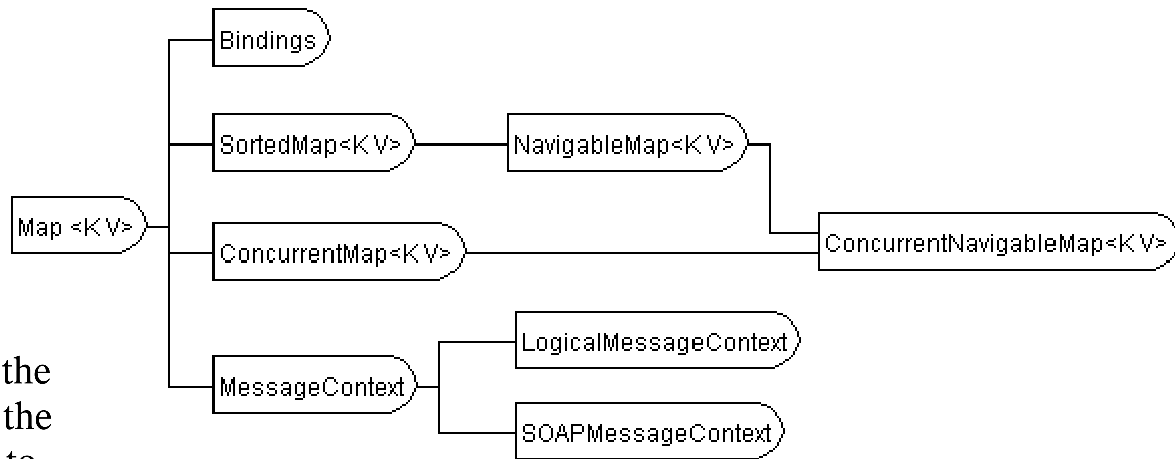
Map Interfaces and Classes



These are the general purpose interfaces and classes in JDK that are part of the Map class and its children.



Collections Interfaces Full Set



This list contains all the interfaces that are in the JDK, and are related to the Map and Collection classes.