CS1092 Object-Oriented Programming with Java

Collective APIness:

Interfaces and Collections

Alan Williams Room 2.107

email: alanw@cs.man.ac.uk

School of Computer Science

Semester 2, 2004-2005

Supporting Material: Interfaces

- these slides
- JTL Book notes: Chapter 19 (later. . .)
- Liang — *Introduction to Java Programming*: Chapter 8.11 (§8.11.1: Interfaces vs. Abstract Classes)
- Savitch — *Absolute Java*: Chpater 13.1

Supporting Material: Collections

- these slides, and JTL Book notes: Chapter 19
- Documentation (local copy):

  `http://www.cs.man.ac.uk/campusonly/jdk1.4/docs/guide/collections/index.html`
- Javadoc for API (local copy):

  `http://www.cs.man.ac.uk/campusonly/jdk1.4/docs/api/index.html`
- Sun Tutorial (excellent!):

  `http://java.sun.com/docs/books/tutorial/collections/`
- Liang — *Introduction to Java Programming*: Chapter 19 (but very poor interface/class diagrams!)
- Savitch — *Absolute Java*: Chapter 15 (but describes **Vector** rather than **ArrayList**)

Outline

- Recap: Arrays
  - storing films
  - sorting
  - extensible/partially-filled
- Representing and processing groups of objects:
  - recall mathematical structures in CS1021
  - set, list, function, ordering
- Interfaces Revisited
- The Collections Framework: part of Java Standard API

Lecture Examples and Lab Exercise

- Example 1: Exam Marks processing
- Example 2: Identifier Indexing — design tool
- Lab: Film Database Deluxe — keyword index generation

Recap: Arrays

variable declaration:	**`String [] myArray`**
create:	**`new String[SIZE];`**
access:	**`aString = myArray[index];`**
update:	**`myArray[index] = "A new String";`**
size:	**`myArray.length`**
sorting:	bubble sort in class **FilmDatabase**
partially-filled arrays:	**`films`** array in class **FilmDatabase**

iteration:

```
for (int i = 0; myArray.length; i++)
  System.out.println(i + ": " + myArray[i])
```

Representing Groups of Objects: CS1021

- sets: $\{e_1, e_2, \ldots, e_n\}$

  empty, isEmpty, member, add, remove
- lists: $[e_1, e_2, \ldots, e_n]$

  append, insert, delete, access/update (via index position)
- relations: $R \subseteq (A_1 \times \cdots \times A_n)$
- functions: $f : Dmn \rightarrow Rng$;

  for $x \in Dmn$, $f(x) = y$, where $y \in Rng$
- (total) ordering on a set S:

  $\forall x, y \in S : x \leq y \vee y \leq x$
- $x = y$ iff $x \leq y \wedge y \leq x$

Example 1: Exam Marks Processing

- Input two files:

  (1) Name/regnumber pairs: unordered

  (2) regnumber/mark pairs: unordered

  Assume: same regnumbers used in each file (?)

- Output: ‘merge’ of elements in (1) and (2) to give Name/mark pairs

  Display in Name and decreasing marks order

- Algorithm:

  (a) first sort (1) and (2)

  (b) now ‘merge’ sorted elements together: create Name/mark pairs

  (c) sort by name

  (d) sort by mark (descending)

A general sorting object would be useful. . .

Recap: CS1081 Film Database Sorting

```
public class FilmDatabase {...
  Film [] selectedFilms;
  ...
  private void sortSelectedFilms()
  { int unsortedLength = numberOfSelectedFilms;
    boolean changedOnThisPass;
    do
    { changedOnThisPass = false;
      for (int i = 0; i < unsortedLength - 1; i++)
        if (selectedFilms[i]
              .compareTo(selectedFilms[i + 1], currentSortMode) > 0)
        {
          swap(i, i+1);
          changedOnThisPass = true;
        }
      unsortedLength--;
    } while (changedOnThisPass) ;
  } // sortSelectedFilms
```

```
public class Film
...
  public int compareTo(Film other, int sortMode)
  {
    switch (sortMode)
    {
      case SORT_BY_TITLE: return title.compareTo(other.title);
      case SORT_BY_DATE:  return dateMade - other.dateMade;
      default:            return 0;
    } // switch
  } // compareTo
```

Simplify **`compareTo()`** for now, just to compare titles:

```
public int compareTo(Film other)
{
  return title.compareTo(other.title);
} // compareTo
```

Records

Now what about sorting name/exam records:

```
// Not the final version!
public class RegRecord
{
  protected final String regnumber;

  public RegRecord(String requiredRegnumber)
  {
    regnumber = requiredRegnumber;
  }

  public String getRegnumber ()
  {
    return regnumber;
  }
} // class RegRecord (not the final version)
```

NameRecord.java (not final version):

```
public class NameRecord extends RegRecord
{
  private final String name;

  public NameRecord(String requiredName, String requiredRegnumber)
  {
    super(requiredRegnumber);
    name = requiredName;
  }

  public String getName()  { return name; }
  public String toString() { return regnumber + ": " + name;  }

  public int compareTo(Object other)
  { return regnumber.compareTo(((RegRecord)other).getRegnumber()); }

} // class NameRecord
```

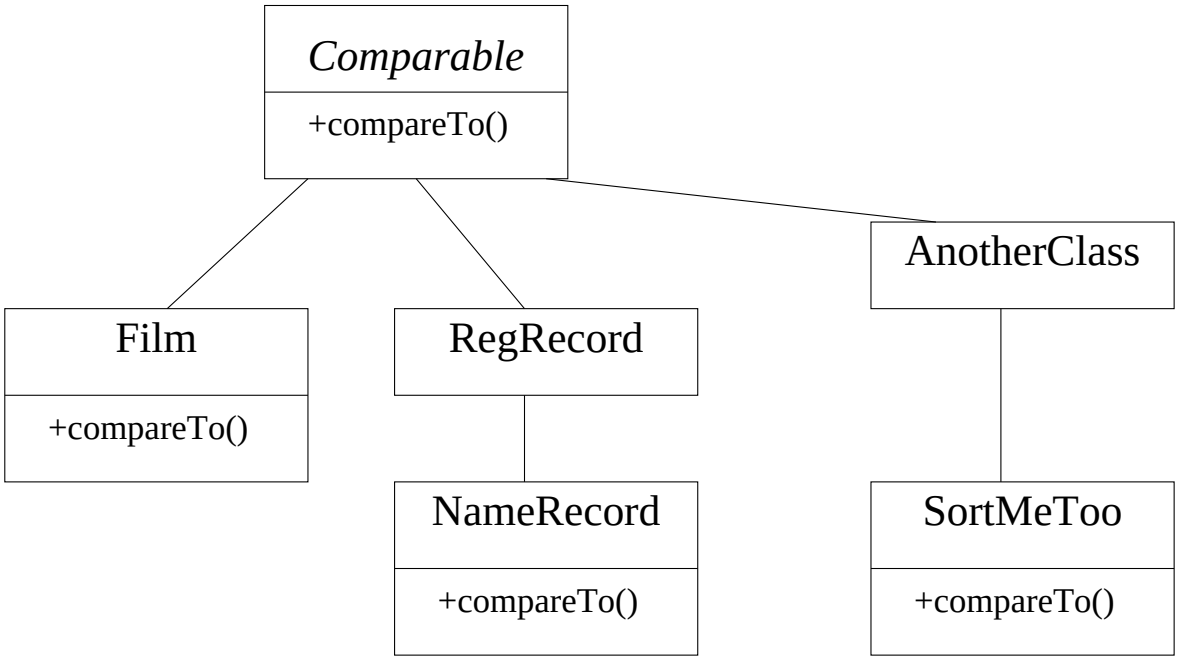
General Purpose Sorting

Modify sorting method so that array holds objects of type **`Comparable`**:

```
Comparable []  selectedItems;
...
private void sortList()  {
  int unsortedLength = numberOfItems;
  boolean changedOnThisPass;
  do
  {
    changedOnThisPass = false;
    for (int i = 0; i < unsortedLength - 1; i++)
      if (selectedItems[i].compareTo(selectedItems[i + 1]) > 0)
      {
        swap{i, i + 1};
        changedOnThisPass = true;
      } // if
    unsortedLength--;
  } while (changedOnThisPass) ;
} // sortSelectedFilms
```

Possible Class Hierarchies

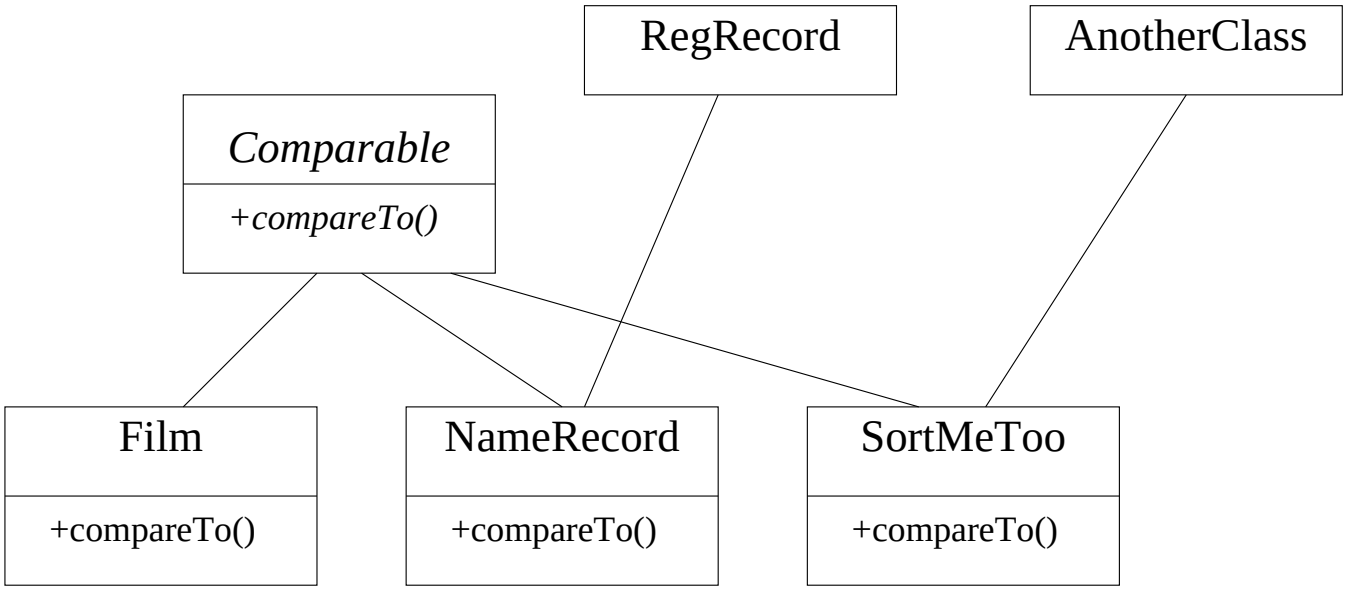
Give all ‘comparable’ objects a common class tree root?:



Not really very useful

>>> Why not?

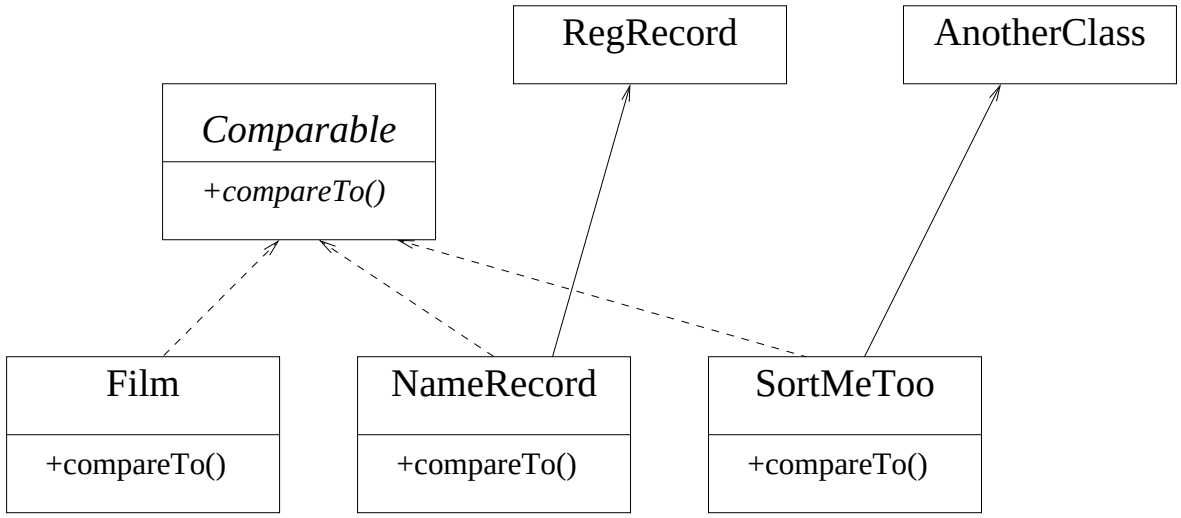
Allow ‘comparable’ objects to inherit from two (or more) super-classes?:



Not allowed in Java!

>>> Why not?

Interfaces



Notice: dashed lines relate Interface and implementation class(es)

Interfaces

Utilised before in GUIs (but not defined):

- similarities to a class
- very similar to a *fully* abstract class:
  - allow only abstract method definitions (i.e. body-less methods)
  - class constants
  - **BUT** no concrete methods (i.e. with bodies)
- interfaces are *types* — you can declare variables/methods of that *type*
- requires *implementing* classes to provide full method definitions for all abstract method specifications
- so . . . code using an object of an interface type knows the interface methods will be defined. . .
- one interface can extend another interface (just like classes)

Defining an Interface

(not seen this before)

```
public interface <Name>
{
   public static final <type> <const> = <value>;
   ...
   public <returntype> <method> (<signature>);
   ...
}
```

No need to declare interface/methods as abstract!

A class can *implement* one or more Interfaces:

```
public class <Name> implements <Inter1>, <Inter2> ...
{
   public <returntype> <method> (<signature>)
   { <body>  }
}
```

(and could extend other classes as well)

Using the Interface type, e.g.:

```
   <Inter1> myObject = new <Name>();

   myObject.<method>(...)
```

(Note <Inter1> in declaration and <Name> in constructor.)

Example: The Comparable Interface

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

That's it!

Small but powerful. . .

Using ***Comparable*** in **NameRecord**:

```
//not final version
public class NameRecord extends RegRecord
                        implements Comparable
{
  ...
  public int compareTo(Object other)
  {
    return regnumber.compareTo(((RegRecord)other).getRegnumber());
  }
  ...
}
```

Note that **`compareTo()`** exactly matches specification in ***Comparable*** definition.

Design of SortedArray

- generalise from **FilmDatabase**
- **`private Comparable [] list;`**
- partially-filled array (as before)
  - **`addItem(Comparable newItem)`**
  - **`Comparable getItem(int index)`**
  - **`int size()`**
  - **`private void increaseSize()`**
  - **`String toString()`**
- generic sort method: **`void sortList()`**

SortedArray.java:

```
public class SortedArray
{
  private static final int INITIAL_SIZE = 2;
  private static final int RESIZE_FACTOR = 2;

  private Comparable [] list;
  private int numberOfItems = 0;

  public SortedArray()
  {
    list = new Comparable[INITIAL_SIZE];
    numberOfItems = 0;
  }
```

```
public void sortList()
{
  int unsortedLength = numberOfItems;

  boolean changedThisPass = false;
  do
  {
    changedThisPass = false;
    for (int i = 0; i < unsortedLength - 1; i++)
    {
      if (list[i].compareTo(list[i+1]) > 0)
      {
        swap(i, i + 1);
        changedThisPass = true;
      }
    }
    unsortedLength--;
  } while (changedThisPass) ;
} // sortList
```

```
private void swap(int from, int to)
{
  Comparable tempValue = list[from];
  list[from] = list[to];
  list[to] = tempValue;
} // swap


public void addItem(Comparable newItem)
{
  if (numberOfItems == list.length)
    increaseSize();

  list[numberOfItems] = newItem;
  numberOfItems++;
}
```

```
private void increaseSize()
{
  Comparable [] newList = new Comparable[list.length * RESIZE_FACTOR];

  System.err.println("Increasing size to " + newList.length);
  for (int i = 0; i< list.length; i++)
    newList[i] = list[i];
  list = newList;
}
```

```
  public Comparable getItem(int i)  { return list[i];  }

  public int size()  { return numberOfItems;  }

  public String toString ()
  {
    String resultString = "";

    for (int i = 0; i < numberOfItems; i++)
    {
      resultString += list[i] + "\n";
    }
    return resultString;
  } // toString
} // class SortedArray
```

Application to Exam Marks Processing: Version 1

- Solve a simpler problem:

  read in unordered name/regnumber pairs file and sort into regnumber-order
- use text file I/O (previous lectures!)
- abstract **Record** class
- class **RegRecord**:
  - holds **`regnumber`**
  - implements ***Comparable*** interface
- use generic array sorter: **SortedArray**

  (just seen this)

RegRecord.java:

```
public class RegRecord extends Record
                       implements Comparable
{
  protected final String regnumber;

  public RegRecord(String requiredRegnumber)
    { regnumber = requiredRegnumber; }

  public String getRegnumber ()  { return regnumber;  }

  public int compareTo(Object other)
  {
    return regnumber.compareTo(((RegRecord)other).getRegnumber());
  }
} // class RegRecord
```

NameRecord.java (final version):
(**`compareTo()`** is now in **RegRecord**)

```
public class NameRecord extends RegRecord
{
  private final String name;

  public NameRecord(String requiredName, String requiredRegnumber)
  {
    super(requiredRegnumber);
    name = requiredName;
  }

  public String getName() { return name; }
  public String toString()  {  return regnumber + ": " + name;  }
} // class NameRecord
```

Test Class: TestSortedNameRecord

Edited highlights:

```
SortedArray recordList = new SortedArray();
...
   String [] words = nextLine.split("[ ]+");
   RegRecord record;
   record = new NameRecord(words[0], words[1]);
   recordList.addItem(record);
...
recordList.sortList();
...
System.out.println("\nSorted Name Records:\n");
System.out.println("" + recordList);
```

TestSortedNameRecord.java

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

public class TestSortedNameRecord
{
  public static void main(String [] args)
  {
    if (args.length != 1)
    {
      System.err.println("Need a filename argument\n"
         + "Usage: TestSortedNameRecord <namefile>");
      System.exit(1);
    }

    String fileName = args[0];
    SortedArray recordList = new SortedArray();
```

```
try
{
  BufferedReader inputFile = new BufferedReader(new FileReader(fileName));
  String nextLine;

  while ((nextLine = inputFile.readLine()) != null)
  {
    String [] words = nextLine.split("[ ]+");
    RegRecord record;

    record = new NameRecord(words[0], words[1]);

    recordList.addItem(record);
  }
  inputFile.close();
}
catch (Exception e)
{
  System.err.println("Error reading " + fileName + ": " + e.getMessage());
}
```

```
    System.out.println("\nUnSorted  Name Records:\n");
    System.out.println("" + recordList);

    recordList.sortList();

    System.out.println("\nSorted Name Records:\n");
    System.out.println("" + recordList);
  }
}
```

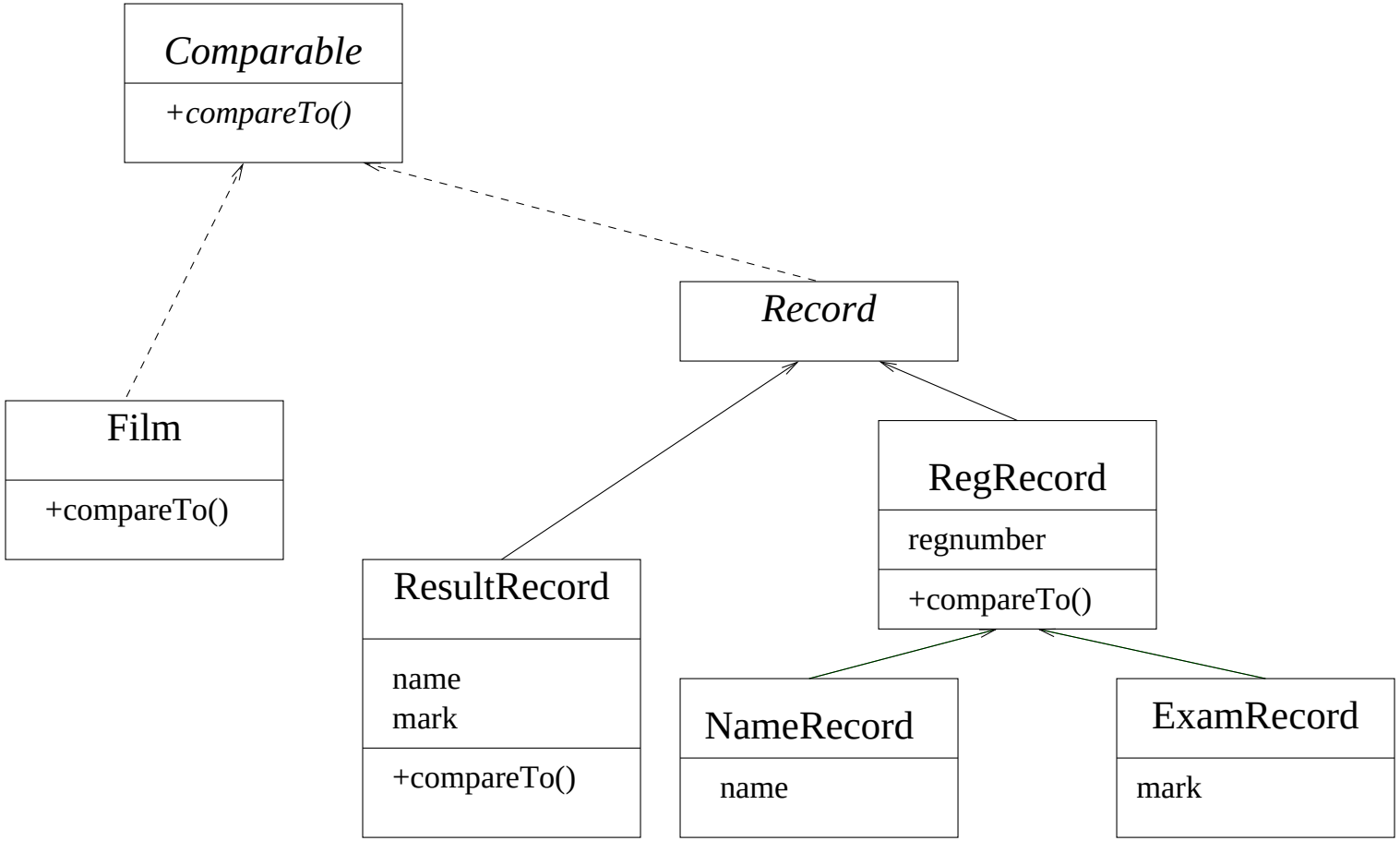
Test Results

```
alanw> java TestSortedNameRecord namefile.txt
Increasing size to 4
Increasing size to 8
Increasing size to 16

UnSorted  Name Records:     Sorted Name Records:
010: Aesop                  001: Alphonse
005: Thiensville            002: Prescott
009: Reeves                 003: Sardinia
008: Amelia                 004: Reid
003: Sardinia               005: Thiensville
007: Judson                 006: Dan
002: Prescott               007: Judson
001: Alphonse               008: Amelia
006: Dan                    009: Reeves
004: Reid                   010: Aesop
```

Exam Marks Processing: Version 2

Complete Design of **Record** classes:



Record.java:

```
public abstract class Record implements Comparable
{
}
```

ResultRecord.java:

```
public class ResultRecord extends Record
{
  private final String name;
  private final String mark;

  public static final int SORT_BY_NAME = 0;
  public static final int SORT_BY_MARK = 1;

  private static  int SORT_MODE = SORT_BY_NAME;

  public ResultRecord(String requiredName, String requiredMark)
  {
    name = requiredName;
    mark = requiredMark;
  }
```

```
public ResultRecord(NameRecord nameRec, ExamRecord examRec)
       throws ResultException
{
  this(nameRec.getName(), examRec.getMark());
  if (! nameRec.getRegnumber().equals(examRec.getRegnumber()))
    throw new ResultException("Reg Numbers Different");
}

public String getName() { return name; }
public String getMark() { return mark; }

public static void sortByMark() { SORT_MODE = SORT_BY_MARK; }
public static void sortByName() { SORT_MODE = SORT_BY_NAME; }
```

```
public int compareTo(Object other)
{
  ResultRecord otherRecord = (ResultRecord)other;
  switch (SORT_MODE)
  {
    case SORT_BY_NAME:
      return name.compareTo(otherRecord.getName());
    case SORT_BY_MARK:
      Integer otherMark = Integer.decode(otherRecord.getMark());
      Integer thisMark = Integer.decode(mark);
      return otherMark.compareTo(thisMark);
    default:
      return name.compareTo(otherRecord.getName());
  }
} // compareTo
```

```
  public String toString()
  {
    switch (SORT_MODE)
    {
      case SORT_BY_NAME: return name + ": " + mark;
      case SORT_BY_MARK: return mark + ": " + name;
      default: return name + ": " + mark;
    }
  } // toString
} // class ResultRecord
```

Putting It All Together

Highlights of **TestSortedArray.java**:

```
private static SortedArray readRecordFile(String fileName, int recordType)
{  ...
      Record record;
      ...
      case NAME_RECORD:
        record = new NameRecord(words[0], words[1]);
        break;
      ...
      recordList.addItem(record);
      ...
  recordList.sortList();
  return recordList;
} // readRecordFile
```

```
public static void main(String [] args) {...
  SortedArray nameList = readRecordFile(args[0], NAME_RECORD);
  SortedArray examList = readRecordFile(args[1], EXAM_RECORD);
  SortedArray resultList = new SortedArray();

  for (int i = 0; i < examList.size(); i++)  {
    NameRecord nameRec = (NameRecord)nameList.getItem(i);
    ExamRecord examRec = (ExamRecord)examList.getItem(i);

    try
    { resultList.addItem(new ResultRecord(nameRec, examRec)); }
    catch (ResultException e)
    { ... }
  }
```

```
    ResultRecord.sortByName();
    resultList.sortList();
    ...
    ResultRecord.sortByMark();
    resultList.sortList();
  } // main
```

Not an ideal way to create different orderings.

>>> Use **TreeSort(Comparator C)** — see later

TestSortedArray.java:

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

public class TestSortedArray
{
  private static final int NAME_RECORD = 0;
  private static final int EXAM_RECORD = 1;

  private static SortedArray readRecordFile(String fileName, int recordType)
  {
    SortedArray recordList = new SortedArray();

    try
    {
      BufferedReader inputFile = new BufferedReader(new FileReader(fileName));
      String nextLine;

      while ((nextLine = inputFile.readLine()) != null)
```

```
  {
    String [] words = nextLine.split("[ ]+");
    Record record;

    switch (recordType)
    {
      case NAME_RECORD:
        record = new NameRecord(words[0], words[1]);
        break;
      case EXAM_RECORD:
        record = new ExamRecord(words[0], words[1]);
        break;
      default:
        record = new NameRecord(words[0], words[1]);
    }

    recordList.addItem(record);
  }
  inputFile.close();
}
catch (Exception e)
```

```
  {
    System.err.println("Error reading " + fileName
       + ": " + e.getMessage());
  }

  System.out.println("\nUnSorted  list:\n");
  System.out.println("" + recordList);

  recordList.sortList();

  System.out.println("\nSorted list:\n");
  System.out.println("" + recordList);
  return recordList;
}

public static void main(String [] args)
{
  if (args.length != 2)
  {
    System.err.println("Need two filename arguments"
      + "Usage: SortedArray <namefile> <examfile>");
```

```
  System.exit(1);
}

System.out.println("Name Records:");
SortedArray nameList = readRecordFile(args[0], NAME_RECORD);

System.out.println("Exam Records:");
SortedArray examList = readRecordFile(args[1], EXAM_RECORD);

SortedArray resultList = new SortedArray();

for (int i = 0; i < examList.size(); i++)
{
  NameRecord nameRec = (NameRecord)nameList.getItem(i);
  ExamRecord examRec = (ExamRecord)examList.getItem(i);

  try
  { resultList.addItem(new ResultRecord(nameRec, examRec)); }
  catch (ResultException e)
  {
    System.err.println(e + "\n>> " + nameRec + "\n>> " + examRec);
```

```
        System.exit(1);
      }
    }
    System.out.println("\nUnSorted Result list:\n");
    System.out.println("" + resultList);

    ResultRecord.sortByName();
    resultList.sortList();

    System.out.println("\nSorted Result list (Name Order):\n");
    System.out.println("" + resultList);

    ResultRecord.sortByMark();
    resultList.sortList();

    System.out.println("\nSorted Result list (Mark Order):\n");
    System.out.println("" + resultList);
  }
} // class TestSortedArray
```

Version 2 Results

```
---------------------------------------
Name Records:     |   Exam Records:
UnSorted  list:   |   UnSorted  list:
                  |
010: Aesop        |   006: 7
005: Thiensville  |   002: 36
009: Reeves       |   003: 38
008: Amelia       |   008: 38
003: Sardinia     |   009: 42
007: Judson       |   005: 43
002: Prescott     |   004: 56
001: Alphonse     |   001: 76
006: Dan          |   010: 83
004: Reid         |   007: 95
---------------------------------------
```

```
---------------------------------------
Name Records:     |   Exam Records:
Sorted list:      | Sorted list:
                  |
001: Alphonse     | 001: 76
002: Prescott     | 002: 36
003: Sardinia     | 003: 38
004: Reid         | 004: 56
005: Thiensville  | 005: 43
006: Dan          | 006: 7
007: Judson       | 007: 95
008: Amelia       | 008: 38
009: Reeves       | 009: 42
010: Aesop        | 010: 83
---------------------------------------
```

```
-------------------------------------------------------
UnSorted Result list:|  Sorted Result list (Name Order):
                     |
Alphonse: 76         |  Aesop: 83
Prescott: 36         |  Alphonse: 76
Sardinia: 38         |  Amelia: 38
Reid: 56             |  Dan: 7
Thiensville: 43      |  Judson: 95
Dan: 7               |  Prescott: 36
Judson: 95           |  Reeves: 42
Amelia: 38           |  Reid: 56
Reeves: 42           |  Sardinia: 38
Aesop: 83            |  Thiensville: 43
                     |
-------------------------------------------------------
```

```
Sorted Result list (Mark Order):

95: Judson
83: Aesop
76: Alphonse
56: Reid
43: Thiensville
42: Reeves
38: Amelia
38: Sardinia
36: Prescott
7: Dan
```

Related Interfaces

- ***Iterator***: see below
- ***Comparator***:

  >>> Look up its definition

  >>> Use with **TreeMap(Comparator c)** and **TreeSet(Comparator c)**
- ***Cloneable***: copy complex objects
- ***ActionListener*** etc: already seen!! GUIs
- ***Serializable***: a different style of use for Interfaces
  - hinted at in File I/O
  - a *marker* interface — contains no method specifications!

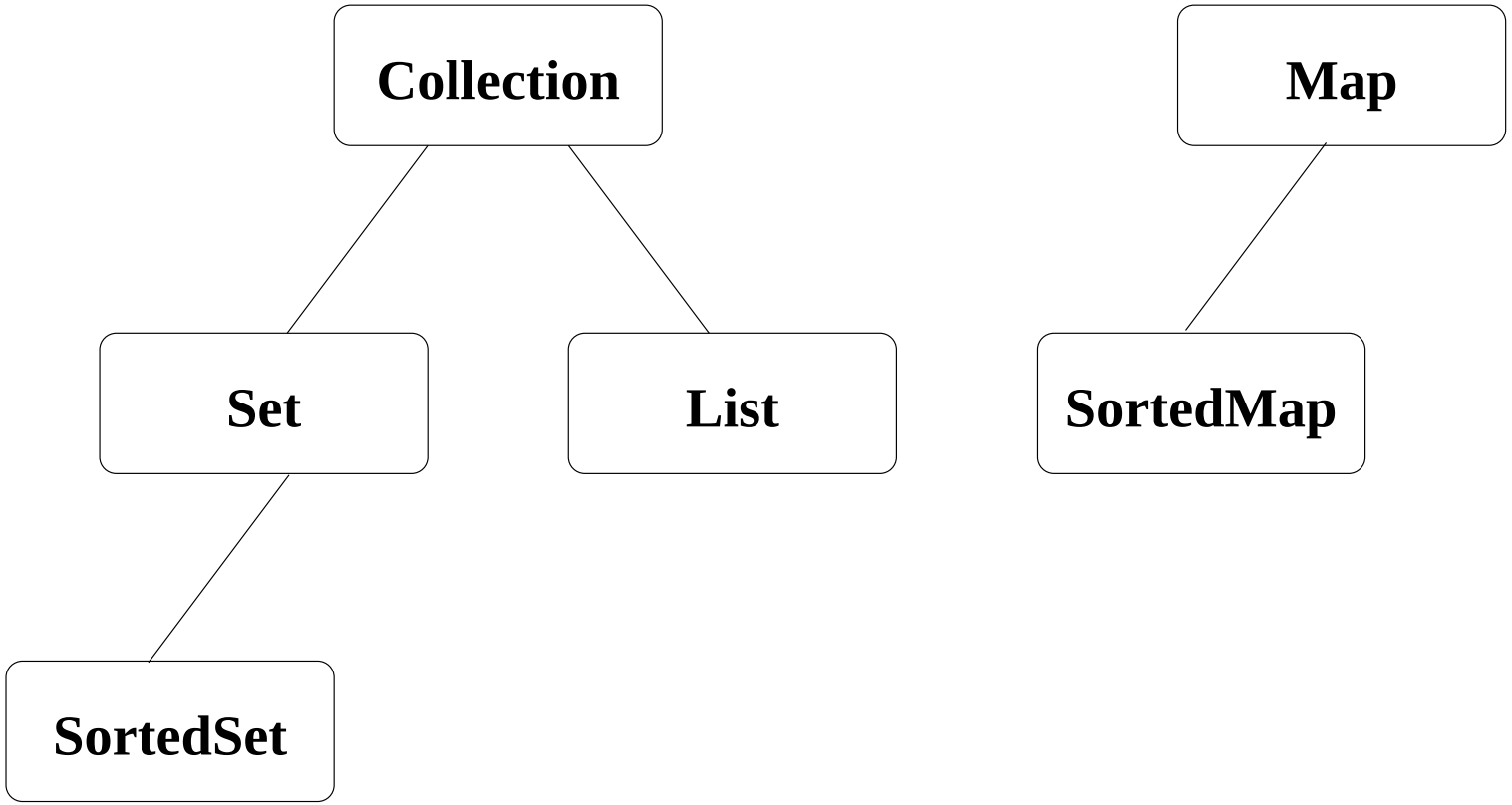
The Collections Framework

- groups elements into a single unit.
- store
- retrieve
- manipulate
- move data as a unit from one object/method to another

(see Sun Collections Tutorial)

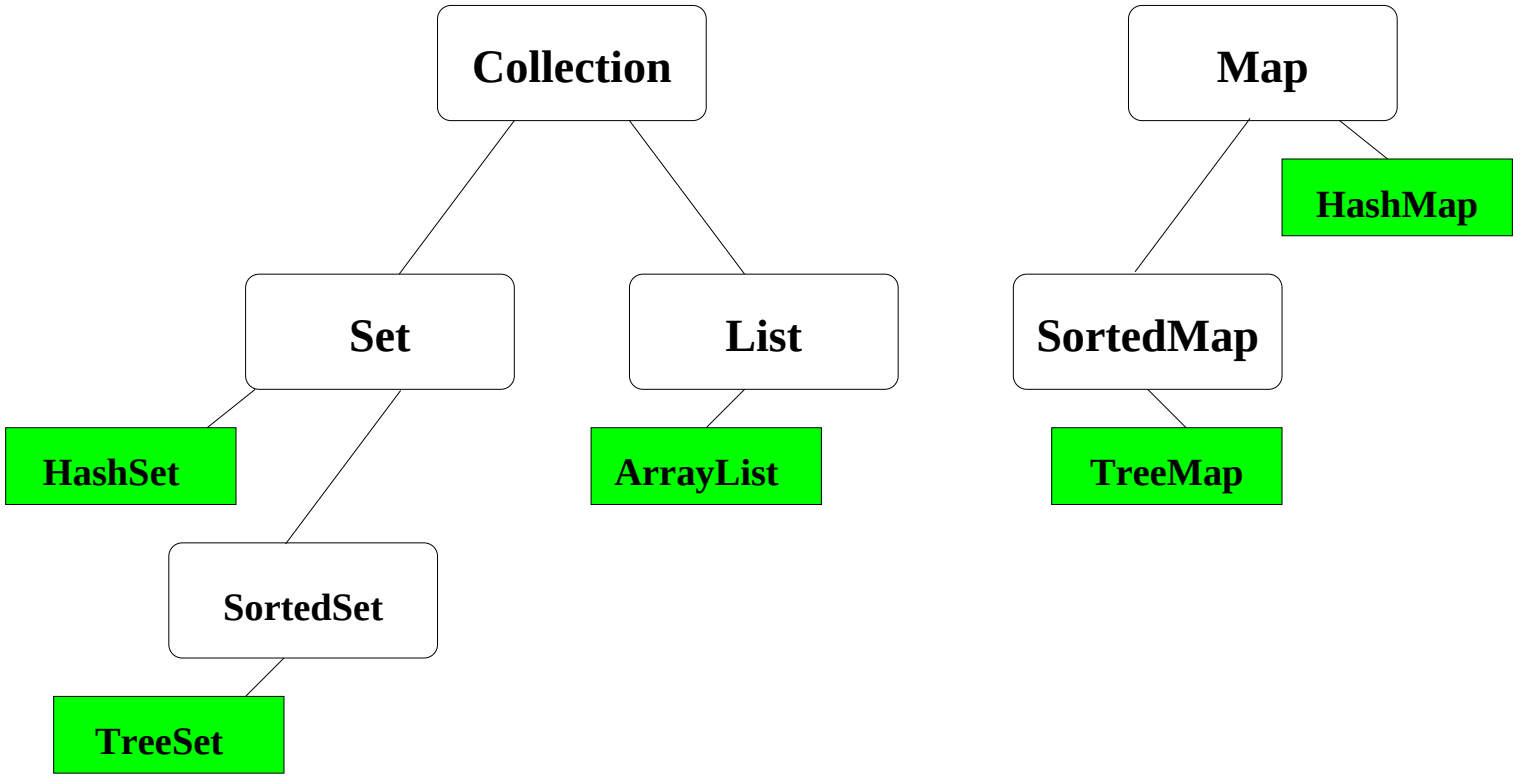
The Collection and Map Interfaces

Allow uniform manipulation of collections/maps:



(see Sun Collections Tutorial)

Some Implementations of the *Collection* Interface



There are more. . .

Interface	Implementations		
	Unsorted	Sorted	In-Order
Set	HashSet	TreeSet	
List			ArrayList
Map	HashMap	TreeMap	

See Sun Java Collections documentation and Tutorial.

e.g. What about **LinkedHashMap**?!

Some Methods in *Collection*

- **`void clear()`**: to remove all elements
- **`boolean isEmpty()`**: true if collection has no elements
- **`boolean add(Object o)`**: add **`o`** (possible duplicates)
- **`boolean remove(Object o)`**: to remove one (or more) elements equal to **`o`**; return true if one exists
- **`boolean contains(Object o)`**: true if an element equals **`o`**
- **`int size()`**: number of elements in collection
- **`Iterator iterator()`**: return an *Iterator* object
- **`Object [] toArray()`**: return an array containing all elements
- **`Object [] toArray(Object [] o)`**: return an array of same type as argument array **`o`**

>>> There are others — see API documentation!

Related Interfaces

Comparable (seen this!)

- **`int compareTo(Object o)`**:
- throws **ClassCastException** if type of **`o`** is incompatible
- consistency with **`equals()`** is required:

  `x.compareTo(o) == 0` iff **`x.equals(y)`** is **`true`**

`x.equals(y)` must be:

- reflexive: `x.equals(x)` is `true`
- symmetric: `x.equals(y)` is `true` iff `y.equals(x)` is `true`
- transitive: if `x.equals(y)` is `true` and `y.equals(z)` is `true` then `x.equals(z)` is `true`

Remember CS1021...

Comparator:

- an implementation should provide a *total ordering* (CS1021!)
- **`int compare(Object x, Object y)`**
- **`boolean equals(Object x, Object y)`**
- consistency with **`equals()`** required
- allows orderings of collection elements different to that given by **`compareTo()`** (the *natural ordering*)

Iterator:

- sequential view and access to elements in ***Collection***
- also modification
- **`boolean hasNext()`**
- **`Object next()`**: note return type
- **`remove()`**: removes, *from Collection*, last element returned by **`next()`**

Set Interface

- provides the familiar mathematical notion of set (CS1021!)
- inherits method specifications from ***Collection***
- no order on objects required
- no repeats: determined via **`equals()`** (and **`hashCode()`**)
- . . . i.e. if **`o.equals(e)`** then **`add(Object o)`** does not add **`o`**, and returns **`false`**

Consistency

What is a **`hashCode()`** for an element????

- required by some implementations (see later)
- it should return an integer!

Also **`compareTo()`**, **`equals()`** and **`hashcode()`** must all be consistently defined

Condition 1 : **`x.equals(y)`** is **`true`** implies that **`x.hashCode() == y.hashCode()`**

Note: implies, *not* iff

Condition 2 : **`x.compareTo(y) == 0`** iff **`x.equals(y)`** is **`true`**

Much more in CS2012 next year!

List Interface

Extra methods specified in interface for indexed insert/access/remove:

- **`void add(int i, Object o)`**: now should *insert* **`o`** at position **`i`** (other elements move up one position)
- **`boolean add(Object o)`**: should append **`o`** to end of list
- **`Object remove(int i)`**: delete and return element at position **`i`** (other elements move down)
- **`void set(int i, Object o)`**: replace element at position **`i`** with **`o`**
- **`Object get(int i)`**: return element at position **`i`**

Very similar to an expanding array (from **FilmDatabase** example in CS1081)

Map Interface

- provides the familiar mathematical function (see CS1021!)
- *maps* keys to values
- inherits methods from **Collection** interface
- no duplicate keys (via **`equals()`**): each key maps to one value (at most)
- **`Object get(Object k)`**: to return the value to which this map maps the key **`k`**
- **`Object put(Object k, Object v)`**: add mapping of key **`k`** to element **`v`**
- **`Object remove(Object k)`**: remove mapping for key **`k`**

- **`boolean containsKey(Object k)`**: to return **`true`** if map contains a key **`k`**
- **`boolean containsValue(Object v)`**: to return **`true`** if at least one keys maps to value **`v`**
- **`Set keySet()`**: to return a *set view* of keys in map
- **`Collection values()`**: to return a *collection view* of values in map
- **`Set entrySet()`**: to return a *set view* of (key,value) pairs (as **Entry** objects)

SortedSet and *SortedMap* Interfaces

- elements will be returned in ascending *natural order*
- e.g. by **Iterator** or **`toArray()`**
- natural order given by **`compareTo()`** . . .
- . . . so Set elements must implement ***Comparable***
- . . . so Map keys must implement ***Comparable***

  (as well as **`equals()`**)

Lots more about underlying implementations in CS2012. . .

Some Related (Standard API) Classes

- **Collections**: polymorphic algorithms on ***Collection*** objects; wrappers (e.g. unmodifiable)
- **Arrays**: various methods for manipulating arrays: sort, search. . .
- **Vector** (already mentioned): similar to **ArrayList**

What is ArrayList?

- implentation of ***List*** interface
- similar to the partially-filled array used in Example 1

  (but extended set of operations)

  >>> Find out about **ArrayList** in Java API documentation

  >>> Re-implement **SortedArray** using **ArrayList**

  >>> Implement ***Set*** interface using **ArrayList**

Choice of Collection

Property	Interface
duplicates?	***List***
maintain input/indexed order?	***List***
set of objects	***Set***
ordering of set elements	***SortedSet***
key/value pairs	***Map***
ordering of map elements	***SortedMap***

Also: consider cost of interface implementation

Example 2: Identifier Indexing

The Problem:

- take list of Java files as command-line arguments
- build index of identifiers for files:
  - Version 1 — relation-based:
    use pair: `(identifier, <filename, linenumber>)`
  - Version 2 — map-based:
    identifier as key, `<filename, linenumber>` pair as entry
- display complete index
- find index entries for identifier key
- sort. . .

Have a common interface to both versions

Version 1: Design and Implementation — Relation-based

```
create index (instance of IdentifierIndex)
for each filename (args)
  open file
  for each line/linenumber:
     split line into words (store in array)
     for each word in line:
        if word is *not* a Java keyword:
           add entry (word, filename, linenumber) to index

test:
  for each element in index
    display element
  retrieve element(s) in index matching identifer
```

Other classes:

- **IndexItem**: represent `<filename, linenumber>` pair
- **KeyIndexItem**: represent (word, **IndexItem**) pair
- **SetIdentifierIndex**: a set of **KeyIndexItem**
  Choice of collection — ***Set*** or ***SortedSet***?
- **SetOfIndexItem**: represent a set of **IndexItem**
- abstract class **SetOfElements**: either **IndexItem** or **KeyIndexItem**

Also need to provide Java keyword ‘stoplist’: use **HashSet** implementation of ***Set***

IndexItem.java:

```
public class IndexItem implements Comparable
{
  private final String fileName;
  private final int lineNumber;

  public IndexItem(String requiredFileName,
                   int requiredLineNumber)
  {
    fileName   = requiredFileName;
    lineNumber = requiredLineNumber;
  }

  public String getFileName() { return fileName; }
  public int getLineNumber()  { return lineNumber; }
```

```
  public int compareTo(Object other)
  { IndexItem otherItem = (IndexItem)other;
    int fileCompare = fileName.compareTo(otherItem.getFileName());

    if (fileCompare != 0)
      return fileCompare;
    else
      return lineNumber - otherItem.getLineNumber();
  }

  public boolean equals(Object other)
  { return ((other instanceof IndexItem) &&
            (this.compareTo(other) == 0));
  }

  public int hashCode()
    { return 29 * fileName.hashCode() + lineNumber; }

  public String toString()
    { return "<" + fileName + ": " + lineNumber + ">"; }
} // class IndexItem
```

KeyIndexItem.java:

```
public class KeyIndexItem implements Comparable
{
  private final String key;
  private final IndexItem item;

  public KeyIndexItem(String requiredKey,
                      IndexItem requiredItem)
  {
    key = requiredKey;
    item = requiredItem;
  }

  public String getKey() { return key; }
  public IndexItem getIndexItem() { return item; }
```

```
public int compareTo(Object other)
{
  KeyIndexItem otherKey = (KeyIndexItem)other;
  int keyCompare = key.compareTo(otherKey.getKey());

  if (keyCompare != 0)
    return keyCompare;
  else
    return item.compareTo(otherKey.getIndexItem());
}
```

```
  public boolean equals(Object other)
  {
    return ((other instanceof KeyIndexItem) &&
            (this.compareTo(other) == 0));
  }

  public int hashCode()
    { return 29 * key.hashCode() + item.hashCode(); }

  public String toString()
    { return "(" + key + ", " + item + ")"; }
} // class KeyIndexItem
```

Implementation Using Sets

IdentifierIndex.java:

```
public interface IdentifierIndex
{
  public SetOfElements keyLookUp(String key);

  public void addIndexItem(String word, IndexItem indexWord);

}
```

SetOfElements.java:

```
public abstract class SetOfElements
{
}
```

SetIdentifierIndex.java:

```
import java.util.Set;
import java.util.HashSet;
import java.util.TreeSet;
import java.util.Iterator;

public class SetIdentifierIndex extends SetOfElements
                                implements IdentifierIndex
{
  //  private final Set index = new TreeSet();
  private final Set index = new HashSet();

  public void addIndexItem(String key, IndexItem itemToAdd)
  {
    KeyIndexItem keyItem = new KeyIndexItem(key, itemToAdd);
    index.add(keyItem);
  }
```

```
public void addKeyIndexItem(KeyIndexItem keyItem)
{ index.add(keyItem); }

public SetOfElements keyLookUp(String key)
{
  Iterator itemList = index.iterator();
  SetIdentifierIndex matchSet = new SetIdentifierIndex();

  while (itemList.hasNext())
  {
    KeyIndexItem item = (KeyIndexItem)(itemList.next());
    if (key.equals(item.getKey()))
      matchSet.addKeyIndexItem(item);
  }
  return matchSet;
}
```

```
  public String toString()
  {
    Iterator itemList = index.iterator();

    String resultString = "";

    while (itemList.hasNext())
    {
      KeyIndexItem item = (KeyIndexItem)(itemList.next());
      resultString += item + "\n";
    }
    return resultString;
  }
} // class SetIdentifierIndex
```

Putting It All Together

Highlights of **TestIndex**:

```
Collection stopList = new HashSet();
...
public static void addKeyWords()
{
  for (int i = 0; i < keyWords.length; i++)
    stopList.add(keyWords[i]);
}
```

```
public static void main(String [] args)  {...
  IdentifierIndex index = new SetIdentifierIndex();

  addKeyWords();
  ...
      while ((nextLine = input.readLine()) != null)
      { String [] words = nextLine.split("\\W+");
        for (int nextWord = 0; nextWord < words.length; nextWord++)
        { String word = words[nextWord];
          if (!stopList.contains(word))
          { IndexItem indexWord = new IndexItem(fileName, lineNumber);
            index.addIndexItem(word, indexWord);
          }
        }
        lineNumber++;
      } ...
  SetOfElements matchingSet = index.getSetOfIndexItem("String");
  System.out.println("Index for \"String\"\n" + matchingSet);
```

TestIndex.java:

```
import java.util.Collection;
import java.util.Set;
import java.util.HashSet;

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


public class TestIndex
{
  public static final String [] keyWords =
    {"", "this", "byte", "final", "new", "throw", "case", "finally", "package",
     "throws", "catch", "float", "null", "long", "for", "protected", "try",
     "class", "true", "static", "do", "instanceof", "strictfp", "readLine",
     "super", "abstract ", "else ", "interface", "switch", "goto",
     "public", "void", "const", "if", "boolean", "enum", "int", "double",
     "return", "volatile", "continue", "implements", "synchronized",
     "break", "extends", "native", "private", "transient", "char", "false",
     "short", "while", "default", "import"};
```

```
Collection stopList = new HashSet();

public static void addKeyWords()
{
  for (int i = 0; i < keyWords.length; i++)
    stopList.add(keyWords[i]);
}

public static void main(String [] args)
{
  IdentifierIndex index = new SetIdentifierIndex();

  Collection stopList = new HashSet();
  addKeyWords(stopList);

  try
  {
    for (int nextArg = 0; nextArg < args.length; nextArg++)
    {
      String fileName = args[nextArg];
```

```
    BufferedReader input = new BufferedReader(new FileReader(fileName));
    String nextLine;
    int lineNumber = 1;
    while ((nextLine = input.readLine()) != null)
    {
      String [] words = nextLine.split("\\W+");
      for (int nextWord = 0; nextWord < words.length; nextWord++)
      {
        String word = words[nextWord];
        if (!stopList.contains(word))
        {
          IndexItem indexWord = new IndexItem(fileName, lineNumber);
          index.addIndexItem(word, indexWord);
        }
      }
      lineNumber++;
    }
  }
}
catch (IOException e)
{
```

```
      System.err.println("IO Exception: " + e);
      System.exit(1);
    }

    System.out.println("Display full index:");
    System.out.println("" + index);

    SetOfElements matchingSet = index.keyLookUp("String");
    System.out.println("Index for \"String\"\n" + matchingSet);
  }
} // TestIndex
```

Version 1: Results Using HashSet

test.txt:

```
1       import java.io.BufferedReader;
2       import java.util.*;
3
4       public class Test
5
6       private final String fileName = new String();
```

test2.txt:

```
1      import java.io.BufferedReader;
2
3      String str = new String();
4
5      String str2;
```

```
alanw> java TestIndex test.txt  test2.txt
Display full index:                        Index for "String"
(java, <test.txt: 2>)                      (String, <test.txt: 6>)
(String, <test.txt: 6>)                    (String, <test2.txt: 3>)
(BufferedReader, <test2.txt: 1>)           (String, <test2.txt: 5>)
(java, <test2.txt: 1>)
(Test, <test.txt: 4>)
(io, <test.txt: 1>)
(str2, <test2.txt: 5>)
(BufferedReader, <test.txt: 1>)
(java, <test.txt: 1>)
(str, <test2.txt: 3>)
(fileName, <test.txt: 6>)
(String, <test2.txt: 3>)
(String, <test2.txt: 5>)
(io, <test2.txt: 1>)
(util, <test.txt: 2>)
```

Version 1: Results Using TreeSet

```
Display full index:
(BufferedReader, <test.txt: 1>)
(BufferedReader, <test2.txt: 1>)
(String, <test.txt: 6>)
(String, <test2.txt: 3>)
(String, <test2.txt: 5>)
(Test, <test.txt: 4>)
(fileName, <test.txt: 6>)
(io, <test.txt: 1>)
(io, <test2.txt: 1>)
(java, <test.txt: 1>)
(java, <test.txt: 2>)
(java, <test2.txt: 1>)
(str, <test2.txt: 3>)
(str2, <test2.txt: 5>)
(util, <test.txt: 2>)
```

```
Index for "String"
(String, <test.txt: 6>)
(String, <test2.txt: 3>)
(String, <test2.txt: 5>)
```

Design and Implementation 2: Map-based

Instead of relation, use a *Map*, e.g.

$$\{(\texttt{java}, (\texttt{test.txt}, 1), (\texttt{java}, (\texttt{test.txt}, 2), (\texttt{java}, (\texttt{test2.txt}, 1)\}$$

becomes:

$$\{\texttt{java} \rightarrow \{(\texttt{test.txt}, 1), (\texttt{test.txt}, 2), (\texttt{test2.txt}, 1)\}\}$$

Implementation Using Map

Modify **SetIdentifierIndex**; keep interface ***IdentifierIndex*** the same!

MapIdentifierIndex.java:

```
import java.util.Map;
import java.util.Set;
import java.util.HashMap;
import java.util.TreeMap;
import java.util.Iterator;

public class MapIdentifierIndex implements IdentifierIndex
{
  // private final Map index = new HashMap();
  private final Map index = new TreeMap();

  public boolean containsKey(String key)
  { return index.containsKey(key);  }
```

```
public void addIndexItem(String key, IndexItem itemToAdd)
  {
    SetOfIndexItem keyValue;
    if (index.containsKey(key))
      keyValue = (SetOfIndexItem)(index.get(key));
    else
    {
       keyValue = new SetOfIndexItem();
       index.put(key, keyValue);
    }
    keyValue.addItem(itemToAdd);
  }

  public SetOfElements keyLookUp(String key)
  {
    if (index.containsKey(key))
      return (SetOfIndexItem)(index.get(key));
    else
      return new SetOfIndexItem();
  }
```

```
  public String toString()
  {
    Set keySet = index.keySet();
    Iterator keyList = keySet.iterator();
    String resultString = "";

    while (keyList.hasNext())
    {
      String key = (String)(keyList.next());
      SetOfIndexItem keyValue = (SetOfIndexItem)(index.get(key));

      resultString += key + ":\n" + keyValue + "\n";
    }
    return resultString;
  }
} // MapIdentifierIndex
```

SetOfIndexItem.java:

```
import java.util.Set;
import java.util.TreeSet;
import java.util.Iterator;

public class SetOfIndexItem extends SetOfElements
{ private final Set setOfIndexItem = new TreeSet();

  public void addItem(IndexItem itemToAdd)
  { setOfIndexItem.add(itemToAdd); }

  public String toString()
  {
    Iterator indexList = setOfIndexItem.iterator();
    String resultString = "";

    while (indexList.hasNext())
      resultString += indexList.next() + "\n";
    return resultString;
  }
} // class SetOfIndex
```

Changes to **TestIndex**: None!

Version 2: Results Using TreeMap

```
alanw-Version2-> java TestIndex test.txt  test2.txt
Display full index:                     Index for "String"
BufferedReader:      io:                <test.txt: 6>
<test.txt: 1>        <test.txt: 1>      <test2.txt: 3>
<test2.txt: 1>       <test2.txt: 1>     <test2.txt: 5>

String:              java:
<test.txt: 6>        <test.txt: 1>
<test2.txt: 3>       <test.txt: 2>
<test2.txt: 5>       <test2.txt: 1>

Test:                str:
<test.txt: 4>        <test2.txt: 3>
fileName:            str2:
<test.txt: 6>        <test2.txt: 5>
                     util:
                     <test.txt: 2>
```

PASS, Tutorial, Lab

- PASS/Tutorial:
  - explore Collections API documentation
  - find out about and use **ArrayList**
  - exercises in using Collections
  - differences between Interfaces, Classes and Abstract Classes
  - Use of ***Comparator*** interface with sorted collections
- Lab:
  - Use the Collections Framework to improve **FilmDatabase**
  - retain public interface
  - build various indices to select films by title/genre/date keywords
  - genre-checking
  - filter common words (‘the’, ‘and’, ‘of’ etc)
  - advanced selection: and/or (optional)
  - user-defined fields (optional)

Summary

- Interfaces: what, when, why
- One interface, several implementations
- General-purpose, reusable data structures and algorithms
- Collections: implementation of set, list, map
- ***Collection***, ***Map*** interfaces
- implementations: **HashSet**, **HashMap**, **ArrayList**, **TreeSet**, **TreeMap**
- associated interfaces: ***Comparable***, ***Comparator***, ***Iterator***
- using standard API classes — don’t be scared!
- read documentation very carefully
- making choices
- interface ‘contracts’ for implementors

HAPPY COLLECTING!