Collections & Sorting

Topics:

- Java Collections Framework
- **`Comparable`** interface
- **`ArrayList`** // 2D arrays
- Sorting Algorithms: *Bubble Sort*, *Insertion Sort*

Chapters 6+13 – "Big Java" book
Chapter 16, Appendix B – "Head First Java" book
Chapters 5+6 – "Introduction to Java Programming" book
Chapter 8 – "Java in a Nutshell" book

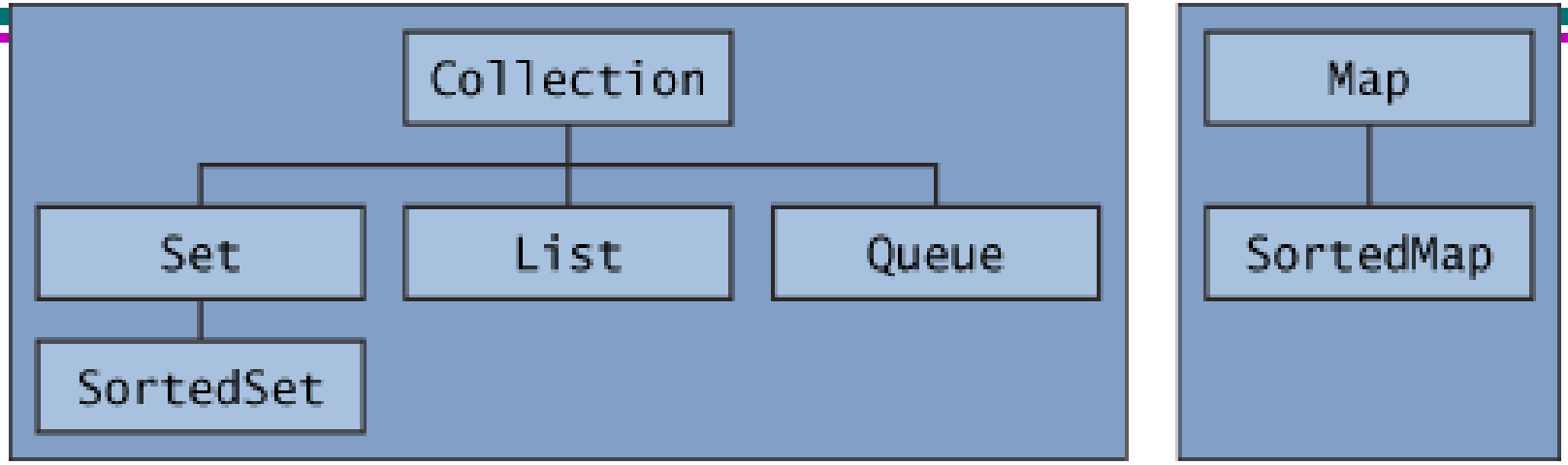
Java's Collection Classes

- Most programs utilise *collections of data* e.g.,
  - a set of users;
  - words in a dictionary.

Example of another data structure?

- Collection in Java: "an object that groups multiple elements into a single unit".
  - Can be used to: store, retrieve, manipulate data.
  - Examples: a poker hand, a mail folder, a telephone directory.
- Java provides several interfaces, implementations and algorithms for handling collections of objects, via its Java Collections Framework (see **`java.util`** package).

Collection Framework: Interfaces



- Set: A collection that contains no duplicate elements; it models the mathematical *set* abstraction.
- List: An ordered collection (also known as a *sequence*). Elements can be accessed by their position in the list, and it is possible to search for elements in the list. Lists allow for duplicate elements.
- Map: An object that maps keys to values. A map does not contain duplicate keys; each key can map to at most one value.

See descriptions of the other interfaces in the Java API.

Choosing a Collection …

- Which collection you choose will depend on many factors e.g.,
  - whether the collection needs to be of fixed size *or* dynamic;
  - whether it has a natural order *or* not;
  - whether we want to insert/delete at arbitrary places, *or* only at the start or end;
  - whether we want to be able to search through a large amount of data very quickly;
  - whether we want random access to elements *or* sequential access;
  - whether we need to maintain key/value pairs.

Collection Framework: Implementations

Interfaces	Implementations				
	Hash table	**Resizable array**	**Tree**	**Linked list**	**Hash table + Linked list**
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

An array that can grow or shrink.

- Array *versus* **`ArrayList`**:
  - An array needs to know its size at time of creation, but an **`ArrayList`** does not,

```
String[] myStringArray = new String[6];
ArrayList<String> myArrayList = new ArrayList<String>();
```

  - To assign an object in a regular array, you must assign it to a specific index, but with an **`ArrayList`** you needn't,

```
anotherList[4] = b;                    anotherArrayList.add(b);
```

Map and HashMap

- The **`Map`** interface associates keys (which are like *indexes*) to elements.
  - It cannot contain duplicate keys.
  - Each key maps to one value only.
- The **`HashMap`** class implements **`Map`** and is efficient for locating a value, as well as inserting and deleting a mapping.
  - Entries are not ordered.
- Example:

Output …

```
> java HashMapTester
map = {mouse=Squeak, cat=Meow,
       guineaPig=Squeak, dog=Woof}
Value for key 'dog' is: Woof
```

```
import java.util.*;
public class HashMapTester {
   public static void main( String[] args ) {
      Map<String, String> petSounds = new HashMap<String, String>();
      petSounds.put("cat", "Meow");  petSounds.put("mouse", "Squeak");
      petSounds.put("dog", "Woof");  petSounds.put("guineaPig", "Squeak");
      System.out.println("map = " + petSounds);
      String val = (String)petSounds.get("dog");
      System.out.println("Value for key 'dog' is: " + val);
   }
}
```

Enumerations

- Enumerations are a Java utility class for holding lists of objects.
  - The idea is that we often encounter pseudo-code design like this:

```
lst is a list of objects
for each item i in lst do {
  process i
}
```

  - Enumerations allow us to code this logic directly.

`Enumeration` interface:

a) introduced before Java 2;

b) *superseeded* by the **`Iterator`** interface.

Example using the `Iterator` Interface

`Iterator` methods …

Result	Method	Description
`b =`	*`it`*`.`**`hasNext`**`()`	`true` if there are more elements for the iterator.
`obj =`	*`it`*`.`**`next`**`()`	Returns the next object. If a generic list is being accessed, the iterator will return something of the list's type. Pre-generic Java iterators always returned type `Object`, so a downcast was usually required.
	`it``.remove()`	Removes the most recent element that was returned by `next`. Not all collections support `delete`. An *UnsupportedOperationException* will be thrown if the collection does not support *remove()*.

```
ArrayList<String> alist = new ArrayList<String>();
// Add Strings to alist

for (Iterator<String> it = alist.iterator(); it.hasNext(); ) {
  String s = it.next(); // No downcasting required.
  System.out.println(s);
}
```

… and things for you to try out!

Arrays (1/2) – Revision

- Arrays in Java are actually *objects*, but they are so commonly used that Java provides language support to simplify their use.
- We've already seen arrays: `public static void main(String[] args)` declares that the one parameter to the **`main`** method is an array of **`String`** objects.
- Arrays can be made of entities of *any* type: objects or base types.
- To *create* an array, we need to use the **`new`** keyword, just as we do with ordinary objects: `int[] intArray;` `intArray = new int[3];`
  - Referencing array elements is the same as in *C*:
    `intArray[0] = 9299;` `intArray[1] = 374;` `intArray[2] = 6473;`
  - Attempting to insert a value at index 3 (or -1) would cause a runtime error **`ArrayIndexOutOfBoundsException`**.

Arrays are not part of the Collections framework.

Arrays (2/2) – Revision

- Usually, we combine the declaration and creation of an array:

```
int[] intArray = new int[3];
```

- Java also supports initialisation of arrays:

```
int[] intArray = {9299, 374, 6473};
String[] strArray = {“hello”, “world”};
System.out.println(sArray[1] + “ “ + sArray[0]);
```

- Array objects have one important instance variable, **`length`**, which determines how long an array is at run-time.

```
public void printArray(Object[] a) {
  for (int i = 0; i < a.length; i++)
    System.out.println(a[i].toString());
}
```

2-dimensional (2D) Arrays

- Java stores a 2D array as an array of arrays, e.g.

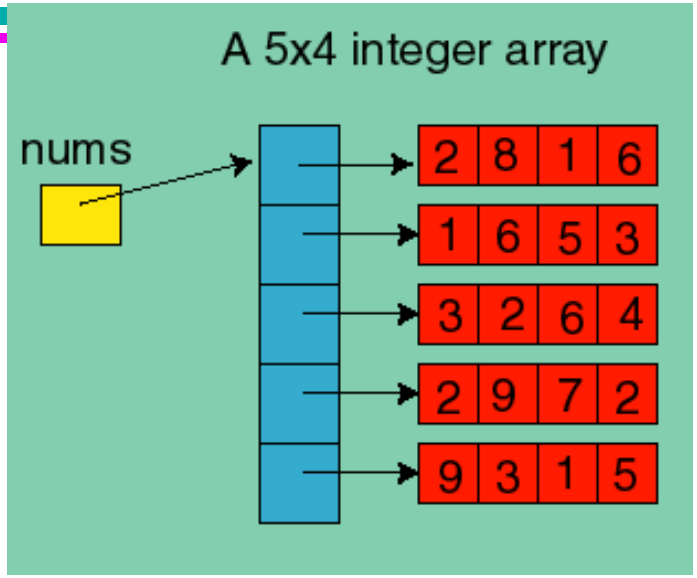
```
int[][] nums = new int[5][4];
```

- When declaring a 2D array:
  - must *always* specify the first dimension

```
nums = new int[][];   // ILLEGAL
```

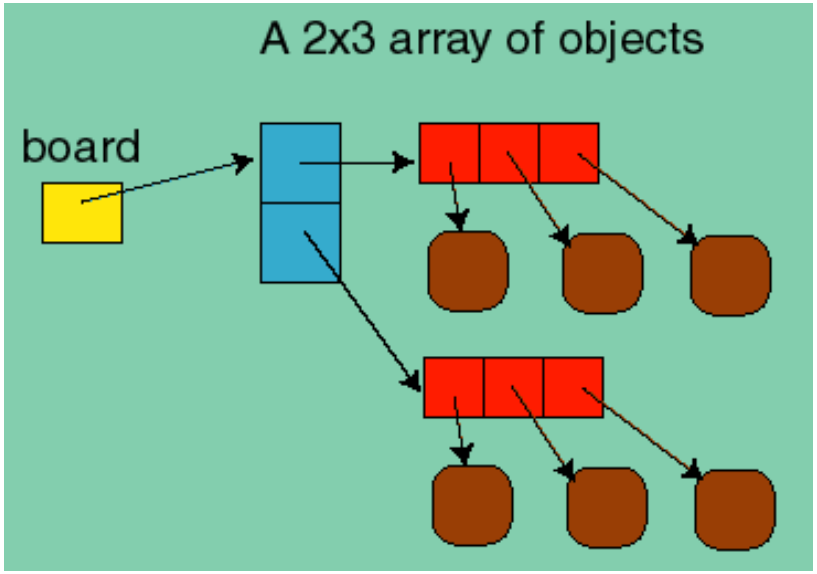
  - do *not* need to specify the second dimension

```
nums = new int[5][];  // OK
nums = new int[5][4]; // OK
```



- A 2D array of objects is an array of an array of references to objects, e.g.

```
Square[][] board = new Square[2][3];
```



Exercise 1 – 2D arrays

- What is the output?

```
public class TwoDArrayTester {
  public static void main(String[] args) {
    char[][] pic = new char[6][6];
    for (int i = 0; i < 6; i++) {
      for (int j = 0; j < 6; j++) {
        if ((i == j) || (i == 0) || (i == 5))
         pic[i][j] = '*';
        else pic[i][j] = '.';
      }
    }
    for (int i = 0; i < 6; i++) {
      for (int j = 0; j < 6; j++)
        System.out.print(pic[i][j]);
      System.out.println();
    }
  }
}
```

Output …

```
******
.*....
..*...
...*..
....*.
******
```

Exercise 2 – 2D ArrayList

- What is the output?

```
import java.util.ArrayList;
public class TwoDArrayListTester {
  public static void main(String[] args) {
    ArrayList<ArrayList<Integer>> topList =
                              new ArrayList<ArrayList<Integer>>();
    for (int i = 0; i < 3; i++)
      topList.add(new ArrayList<Integer>());

    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++)
        topList.get(i).add(new Integer(i+j));
    }
    for (int i = 0; i < 3; i++) {
      for (int j = 0; j < 3; j++)
        System.out.print(topList.get(i).get(j) + " ");
      System.out.println();
    }
  }
}
```

Output …

```
0 1 2
1 2 3
2 3 4
```

Sorting in Java

- Sorting: Common task when programming, e.g. displaying the words in a list in alphabetical order.
  - Algorithms for Sorting:
    - ***Selection*** sort
      1. Find largest number and put it in the last position.
      2. Find next largest number and put it next to last one.
      3. Repeat until finished.
    - ***Insertion*** sort: List of values is sorted by inserting (repeatedly) an unsorted element into a sorted sublist until the complete list is sorted.
    - ***Bubble*** sort (***next …***)

See https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html for more sorting algorithms (with animations).

Bubble (or *Sinking*) Sort

- **Bubble** sort:
  - Several passes are made through the array.
  - Each time, successive adjacent pairs are compared.
    - If pair is in decreasing order, order of values is swapped.
    - Otherwise, move on to next pair.
  - **Algorithm**:

```
boolean changed = true;
do {
  changed = false;
  for (int j=0; j < list.length-1; j++) {
    if (list[j] > list[j+1]) {
      swap list[j] with list[j+1];
      changed = true;
    }
  }
} while (changed);
```

Exercise 3

- Write a sort method that uses the *bubble-sort algorithm*.
  - Use `{5.0, 4.4, 1.9, 2.9, 3.4, 2.9, 3.5}` to test the method.

Homework: Try doing Programming Exercise 5.15 (IJP book).

(*Answer*) Exercise 3

- A possible solution …

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




  }
  void printList(double[] list) {


  }
  // Method bubbleSort() here.
}
```

```
void bubbleSort(double[] list) {




}
```

Comparing objects … (1/2)

- Real objects are often “comparable”, e.g. you (students) are much younger than me! ☺
- When writing OO programs, it may be necessary to compare instances of the same class; once instances are comparable, they can be sorted e.g.
  - Given two **`Employee`** objects, it may be necessary to know which one has been with the company for longer.

- Java defines two ways of comparing objects:
  1. The objects implement the **`Comparable`** interface.
  2. A **`Comparator`** object is used to compare the two objects.
- If the objects are **`Comparable`**, they are said to be sorted by their "natural" order.
- A **`Comparable`** object can only offer one form of sorting.
  - To provide multiple forms of sorting, **`Comparators`** must be used.

`Comparator` is *out of scope* in this course.

Comparing objects … (2/2)

- (Remember?) The **`String`** class has the method **`int compareTo(Object)`**, which returns:
  - **`0`** → if the **`String`**s are equal;
  - **`<0`** → if this object is less than the specified object;
  - **`>0`** → if this object is greater than the specified object.

- The **`Comparable`** interface contains the **`compareTo()`** method.
  - If you wish to provide a natural ordering for your objects, you must implement the **`Comparable`** Interface.
  - Any object which is "comparable" can be compared to another object of the same type.
- There is only one method defined within this interface. So there is only one natural ordering for objects of a given type/class.

Example using Comparable (1/2)

```
public class Employee implements Comparable<Employee> {
  int empID;
  String eName;
  double salary;
  static int i;

  public Employee() {
    empID = i++;
    eName = "unknown";
    salary = 0.0;
  }
  public Employee(String name, double sal) {
    empID = i++;
    eName = name;
    salary = sal;
  }
  public String toString() {
    return "EmpID  = " + empID + "\n" + "Ename  = " + eName + "\n" +
           "Salary = " + salary;
  }
```

```
(rest of the Employee class code)
  public int compareTo(Employee o1) {
    if (this.salary == o1.salary) return 0;
    else if (this.salary > o1.salary) return 1;
    else return -1;
  }
}
```

Aim: To order employees by salary value.

Example using Comparable (2/2)

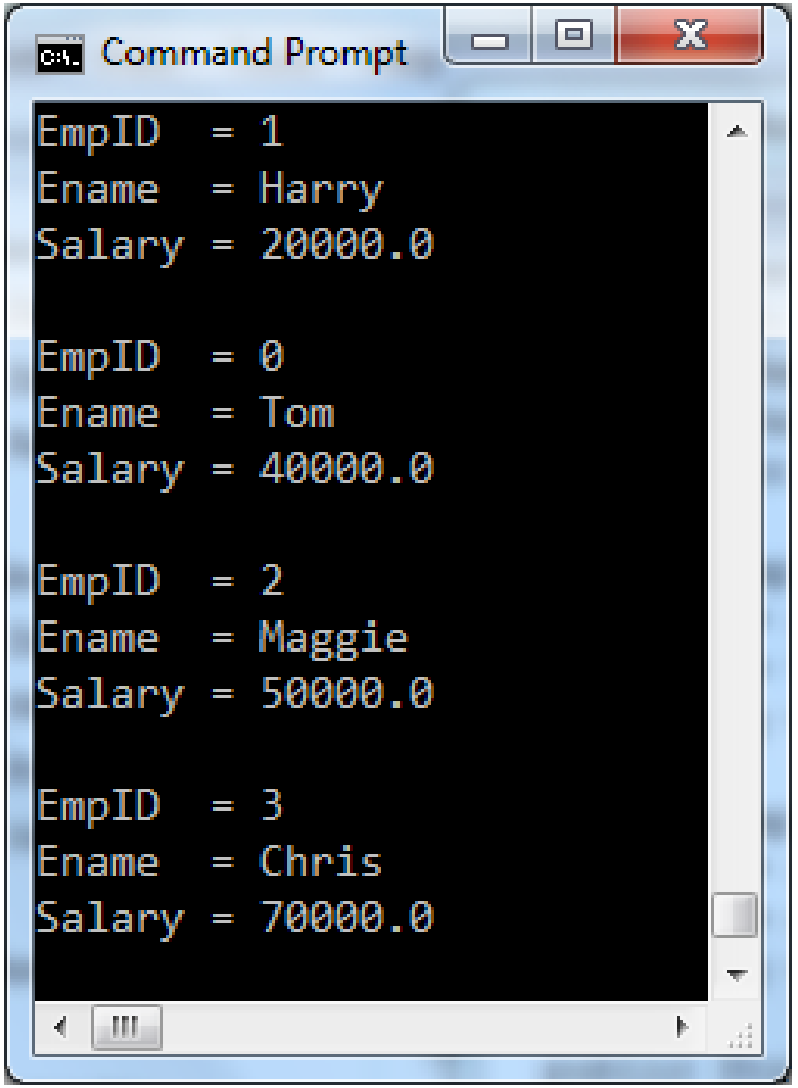
```
import java.util.*;
public class ComparableDemo {
  public static void main(String[] args) {
    List<Employee> ts1 = new ArrayList<Employee>();

    ts1.add(new Employee("Tom", 40000.00));
    ts1.add(new Employee("Harry", 20000.00));
    ts1.add(new Employee("Maggie", 50000.00));
    ts1.add(new Employee("Chris", 70000.00));

    Collections.sort(ts1);
    Iterator <Employee> itr = ts1.iterator();

    while(itr.hasNext()) {
      Object element = itr.next();
      System.out.println(element + "\n");
    }
  }
}
```

Output …



… and things for you to try out!