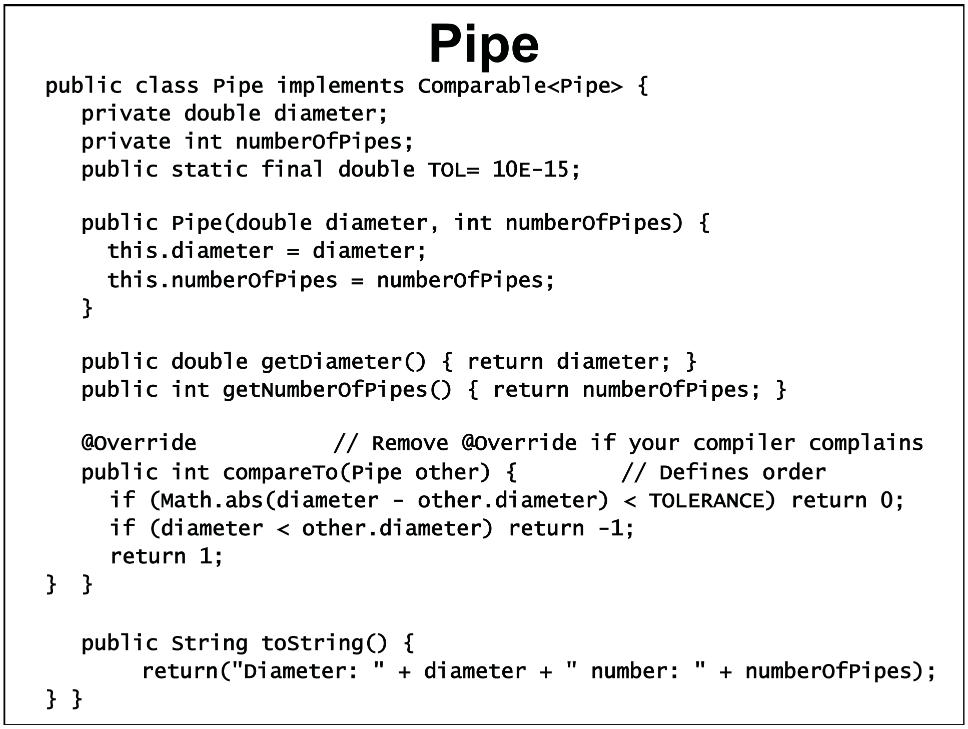
1.00 Lecture 34

Sorting

Reading for next time: Big Java 15.4

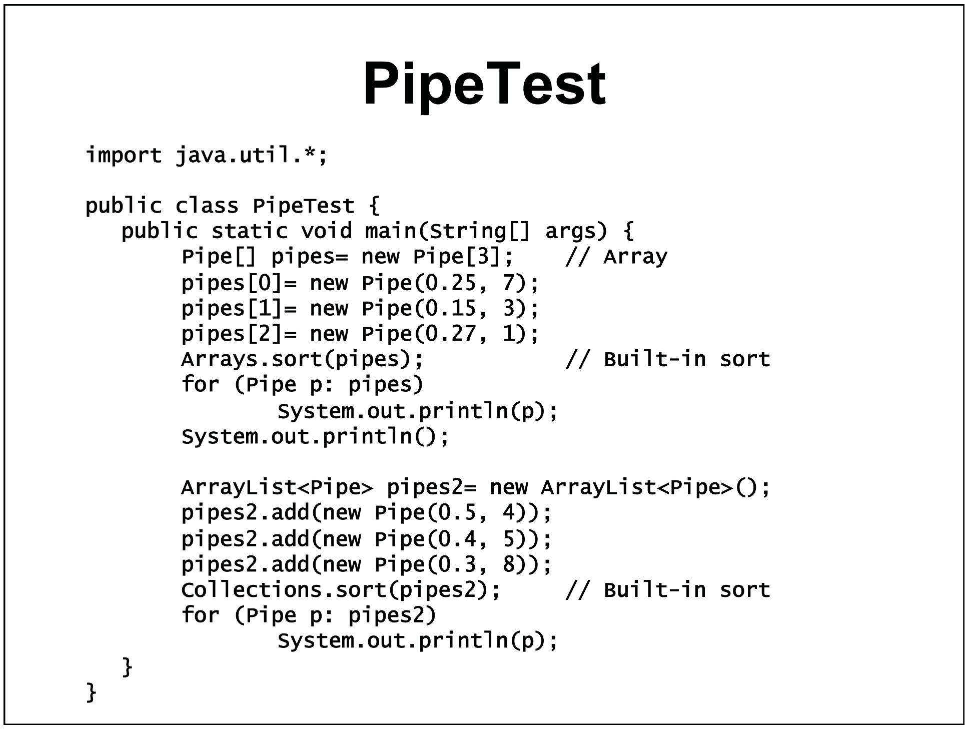
Comparable interface

- Used to define the natural order of objects in a class
  - Supports only a single ordering for the objects
  - Method compareTo() must be implemented
- Used by Java's built-in sort methods for arrays and collections (e.g., ArrayList, LinkedList, Dequeue)
  - Based on mergesort
- We will sort pipes by diameter



Pipe

```
public class Pipe implements Comparable<Pipe> {
   private double diameter;
   private int numberOfPipes;
   public static final double TOL= 10E-15;

   public Pipe(double diameter, int numberOfPipes) {
     this.diameter = diameter;
     this.numberOfPipes = numberOfPipes;
   }

   public double getDiameter() { return diameter; }
   public int getNumberOfPipes() { return numberOfPipes; }

   @Override           // Remove @Override if your compiler complains
   public int compareTo(Pipe other) {        // Defines order
     if (Math.abs(diameter - other.diameter) < TOLERANCE) return 0;
     if (diameter < other.diameter) return -1;
     return 1;
}  }

   public String toString() {
        return("Diameter: " + diameter + " number: " + numberOfPipes);
} }
```



PipeTest

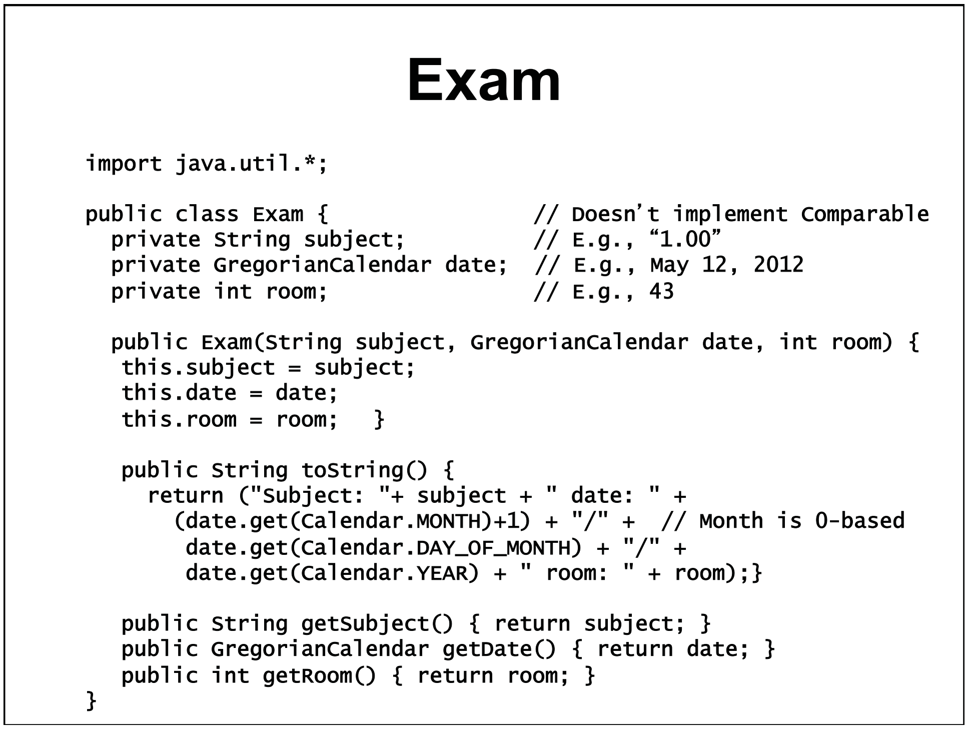
```
import java.util.*;

public class PipeTest {
   public static void main(String[] args) {
        Pipe[] pipes= new Pipe[3];    // Array
        pipes[0]= new Pipe(0.25, 7);
        pipes[1]= new Pipe(0.15, 3);
        pipes[2]= new Pipe(0.27, 1);
        Arrays.sort(pipes);           // Built-in sort
        for (Pipe p: pipes)
                System.out.println(p);
        System.out.println();

        ArrayList<Pipe> pipes2= new ArrayList<Pipe>();
        pipes2.add(new Pipe(0.5, 4));
        pipes2.add(new Pipe(0.4, 5));
        pipes2.add(new Pipe(0.3, 8));
        Collections.sort(pipes2);     // Built-in sort
        for (Pipe p: pipes2)
                System.out.println(p);
   }
}
```

Exercise 1: Comparable

- Modify Pipe and/or PipeTest to sort the pipes by number of pipes, in <u>descending</u> order

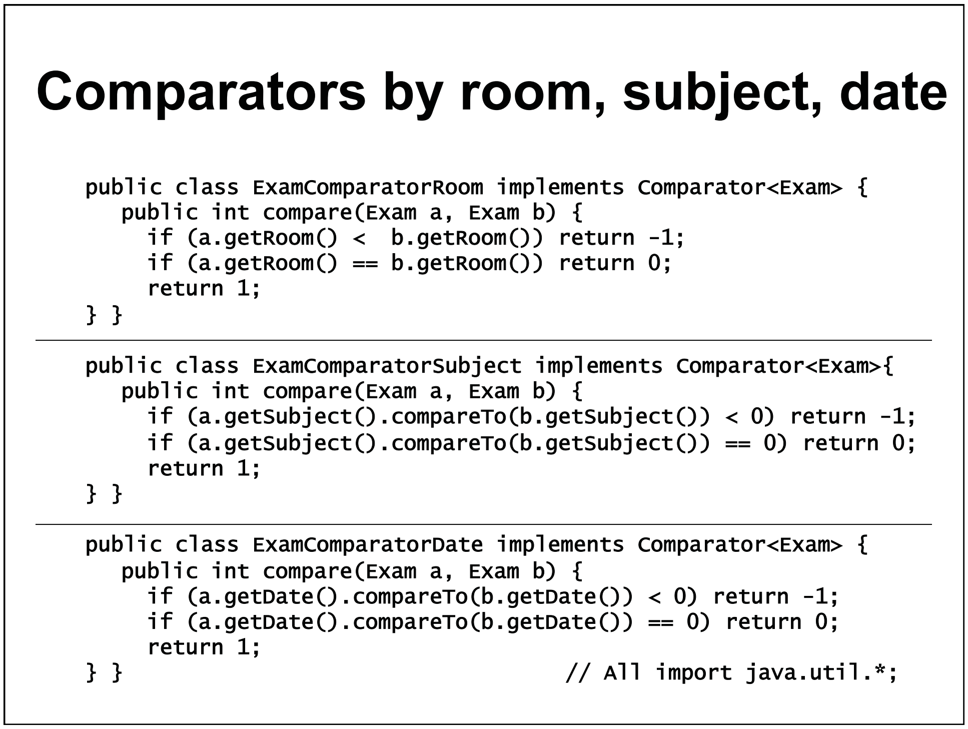
Comparator interface

- Comparators are used:
  - To sort Objects that do not implement Comparable interface
  - When Objects must be sorted in different orders within a program
- Method compare() must be implemented
- We will sort final exams by subject number, date and room number



Exam

```
import java.util.*;

public class Exam {                // Doesn’t implement Comparable
  private String subject;          // E.g., “1.00”
  private GregorianCalendar date;  // E.g., May 12, 2012
  private int room;                // E.g., 43

  public Exam(String subject, GregorianCalendar date, int room) {
   this.subject = subject;
   this.date = date;
   this.room = room;   }

   public String toString() {
     return ("Subject: "+ subject + " date: " +
       (date.get(Calendar.MONTH)+1) + "/" +  // Month is 0-based
        date.get(Calendar.DAY_OF_MONTH) + "/" +
        date.get(Calendar.YEAR) + " room: " + room);}

   public String getSubject() { return subject; }
   public GregorianCalendar getDate() { return date; }
   public int getRoom() { return room; }
}
```

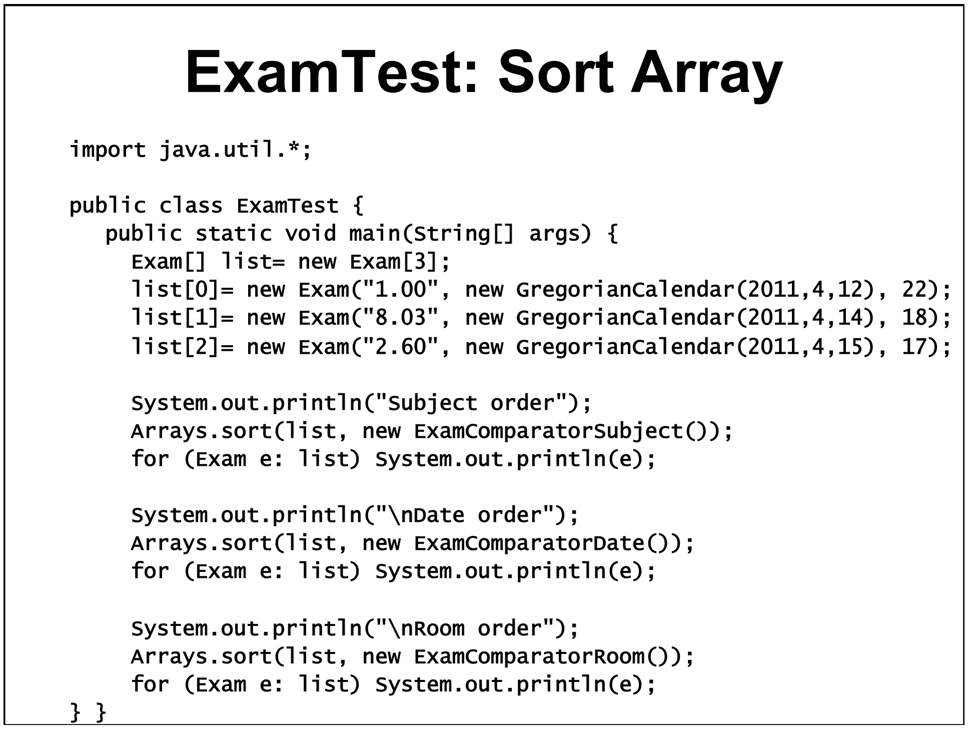


Comparators by room, subject, date

```
public class ExamComparatorRoom implements Comparator<Exam> {
   public int compare(Exam a, Exam b) {
     if (a.getRoom() <  b.getRoom()) return -1;
     if (a.getRoom() == b.getRoom()) return 0;
     return 1;
} }
```

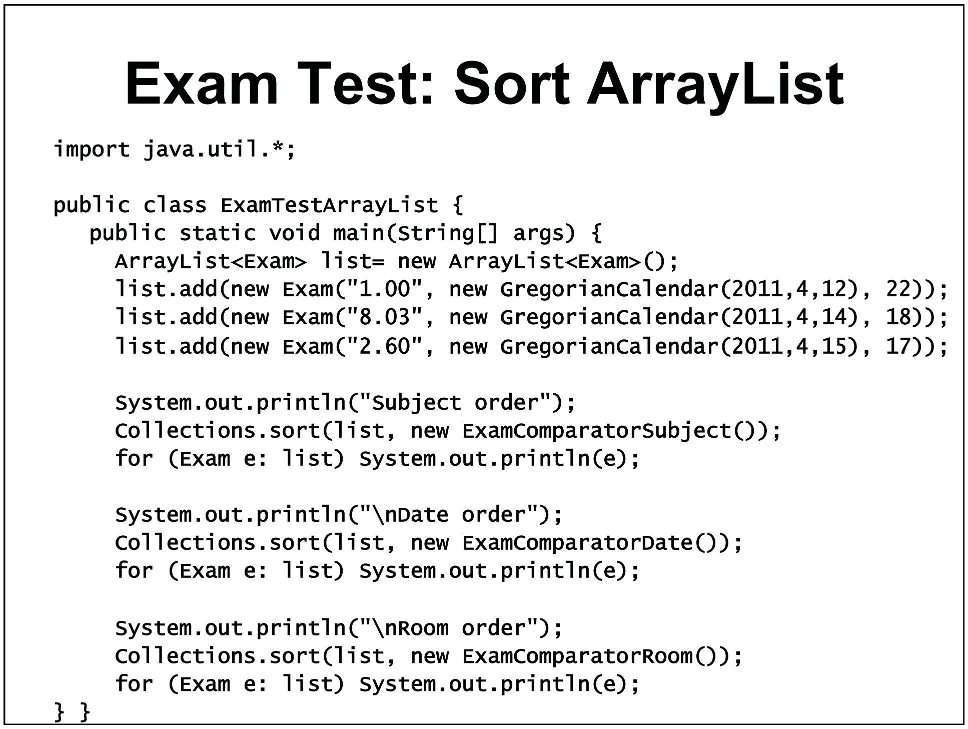
```
public class ExamComparatorSubject implements Comparator<Exam>{
   public int compare(Exam a, Exam b) {
     if (a.getSubject().compareTo(b.getSubject()) < 0) return -1;
     if (a.getSubject().compareTo(b.getSubject()) == 0) return 0;
     return 1;
} }
```

```
public class ExamComparatorDate implements Comparator<Exam> {
   public int compare(Exam a, Exam b) {
     if (a.getDate().compareTo(b.getDate()) < 0) return -1;
     if (a.getDate().compareTo(b.getDate()) == 0) return 0;
     return 1;
} }                                  // All import java.util.*;
```



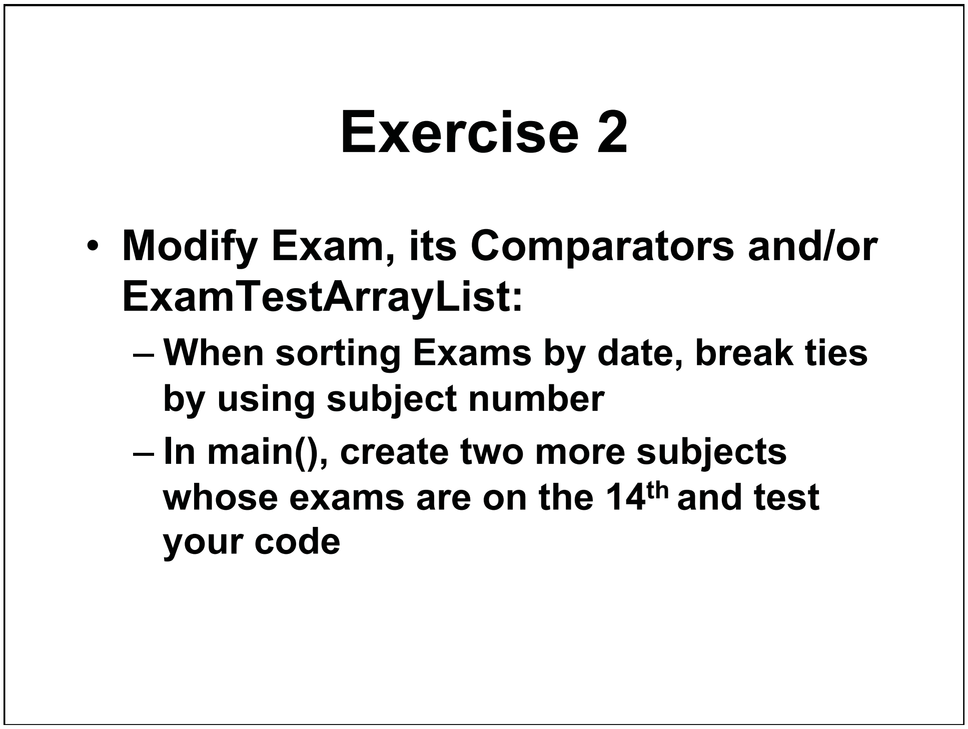
ExamTest: Sort Array

```
import java.util.*;

public class ExamTest {
   public static void main(String[] args) {
     Exam[] list= new Exam[3];
     list[0]= new Exam("1.00", new GregorianCalendar(2011,4,12), 22);
     list[1]= new Exam("8.03", new GregorianCalendar(2011,4,14), 18);
     list[2]= new Exam("2.60", new GregorianCalendar(2011,4,15), 17);

     System.out.println("Subject order");
     Arrays.sort(list, new ExamComparatorSubject());
     for (Exam e: list) System.out.println(e);

     System.out.println("\nDate order");
     Arrays.sort(list, new ExamComparatorDate());
     for (Exam e: list) System.out.println(e);

     System.out.println("\nRoom order");
     Arrays.sort(list, new ExamComparatorRoom());
     for (Exam e: list) System.out.println(e);
} }
```



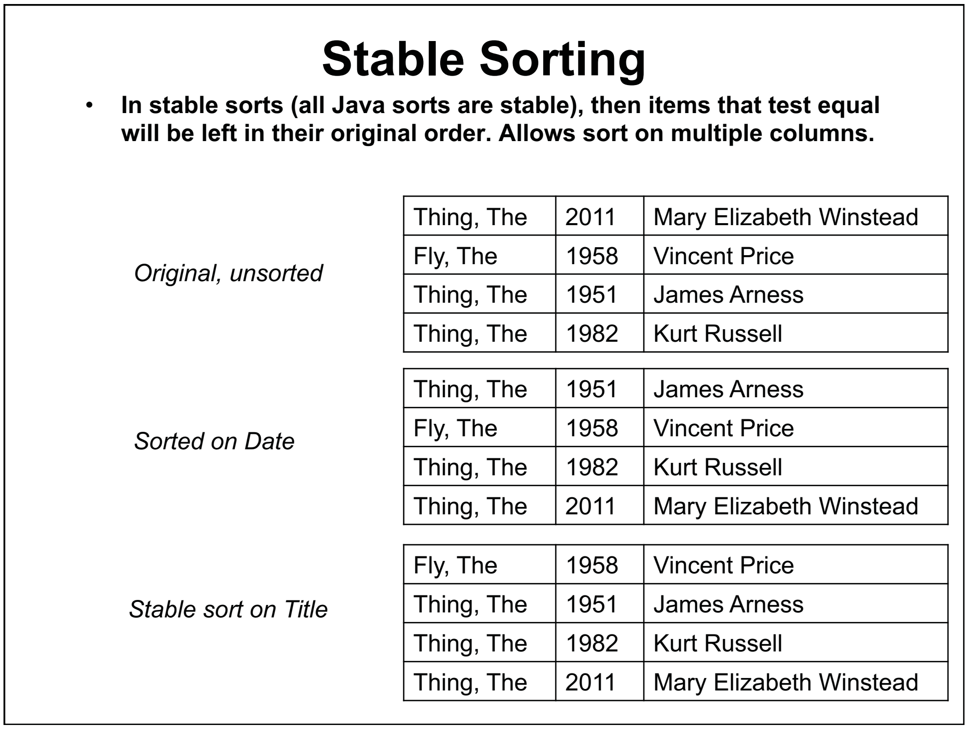
Exam Test: Sort ArrayList

```
import java.util.*;

public class ExamTestArrayList {
   public static void main(String[] args) {
     ArrayList<Exam> list= new ArrayList<Exam>();
     list.add(new Exam("1.00", new GregorianCalendar(2011,4,12), 22));
     list.add(new Exam("8.03", new GregorianCalendar(2011,4,14), 18));
     list.add(new Exam("2.60", new GregorianCalendar(2011,4,15), 17));

     System.out.println("Subject order");
     Collections.sort(list, new ExamComparatorSubject());
     for (Exam e: list) System.out.println(e);

     System.out.println("\nDate order");
     Collections.sort(list, new ExamComparatorDate());
     for (Exam e: list) System.out.println(e);

     System.out.println("\nRoom order");
     Collections.sort(list, new ExamComparatorRoom());
     for (Exam e: list) System.out.println(e);
} }
```



Exercise 2

- Modify Exam, its Comparators and/or ExamTestArrayList:
  - When sorting Exams by date, break ties by using subject number
  - In main(), create two more subjects whose exams are on the 14th and test your code



Stable Sorting

- In stable sorts (all Java sorts are stable), then items that test equal will be left in their original order. Allows sort on multiple columns.

Original, unsorted

Thing, The	2011	Mary Elizabeth Winstead
Fly, The	1958	Vincent Price
Thing, The	1951	James Arness
Thing, The	1982	Kurt Russell

Sorted on Date

Thing, The	1951	James Arness
Fly, The	1958	Vincent Price
Thing, The	1982	Kurt Russell
Thing, The	2011	Mary Elizabeth Winstead

Stable sort on Title

Fly, The	1958	Vincent Price
Thing, The	1951	James Arness
Thing, The	1982	Kurt Russell
Thing, The	2011	Mary Elizabeth Winstead

Applications of sorting

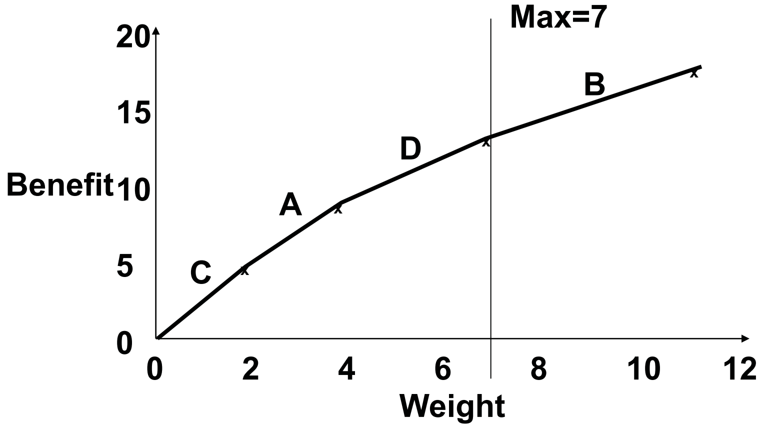
- Business applications are obvious:
  - Reports sorted by customer, department, etc.
- Many technical applications use sorting:
  - Packing (we’ll do a satellite payload exercise)
  - Maintenance, reliability analysis
  - Processor scheduling
  - Graphs (networks): finding paths, spanning trees, etc.
- One estimate is that 25% of all CPU time is spent on sorting

Satellite payload

Experiment	Benefit	Weight	Ben/Wgt
A	4	2	2.0
B	6	4	1.5
C	5	2	2.5
D	5	3	1.7

Assume satellite can carry max weight= 7

Satellite, p.2

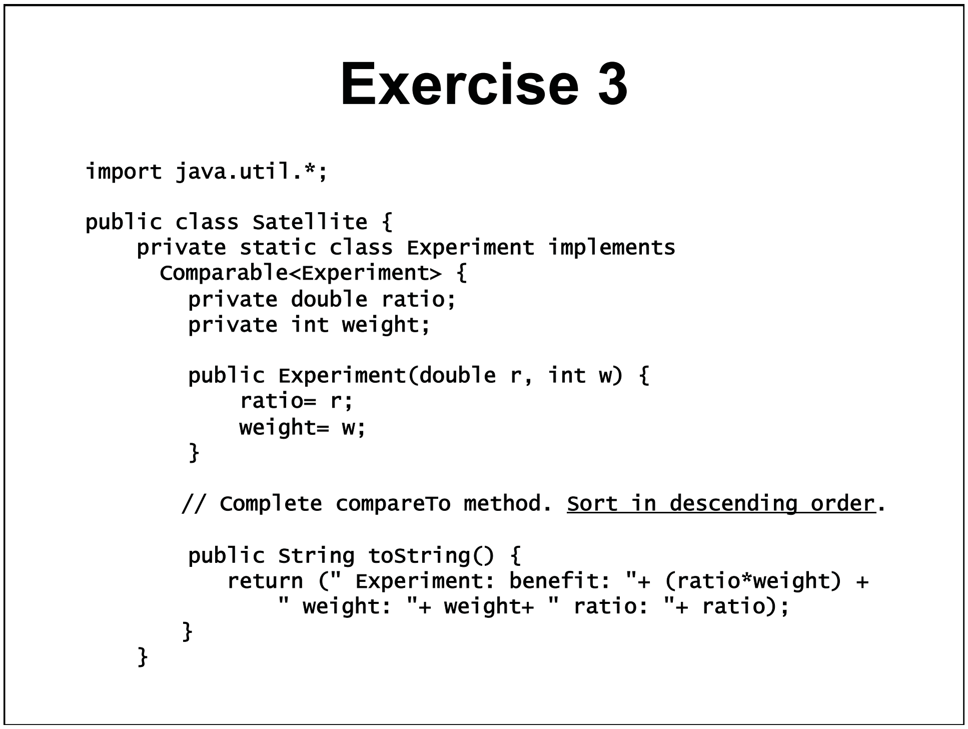


Sort in Benefit/Weight ratio. Maximum derivative gives optimum value.
Gives solution for all maximum weights M. ('Greedy algorithm')

Last item in solution may be fractional. Often this is acceptable.
If not, we use more sophisticated optimization methods

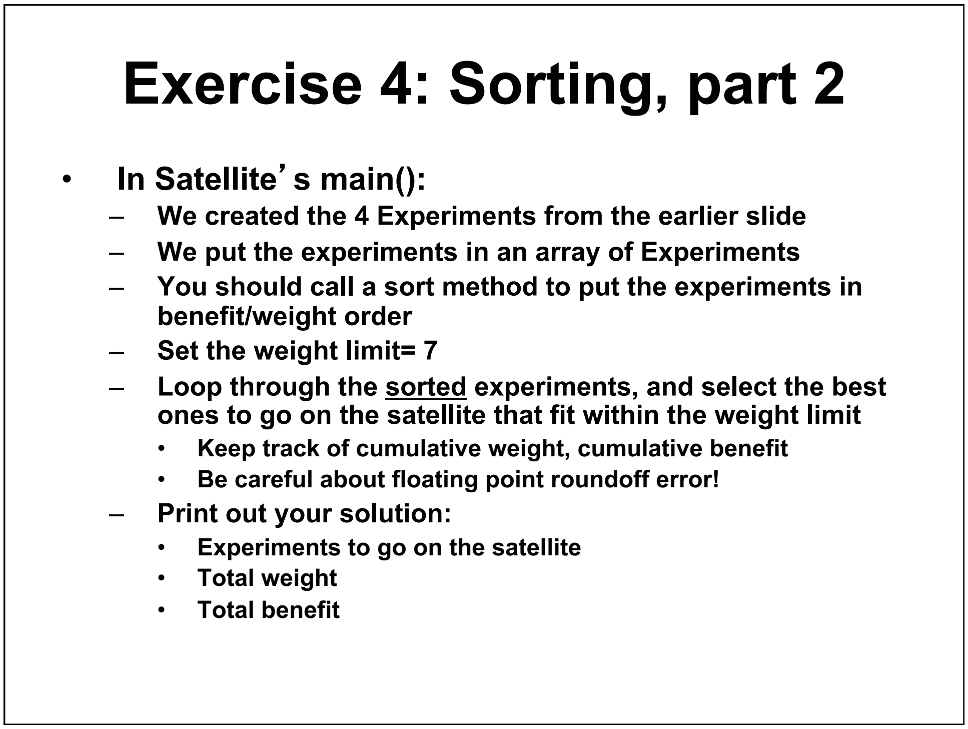
Exercise 3: Sorting

- Finish class Satellite, which selects experiments to be taken aloft. We' ll do it in two steps.
- Create an Experiment static (nested) class to store the benefit/weight ratio (key) and weight (value)
  - Experiment data members: ratio, weight
  - Experiment, to be sortable, must implement Comparable
  - Implement compareTo()
    - We want to sort Experiments in descending order
    - Use the generic (<Experiment>) Comparable interface
- A nested class is like an inner class
  - It uses the static keyword in the class declaration
  - It has no access to its enclosing class' s data
  - It is primarily used for classes that just store data for the convenience of the outer (enclosing) class



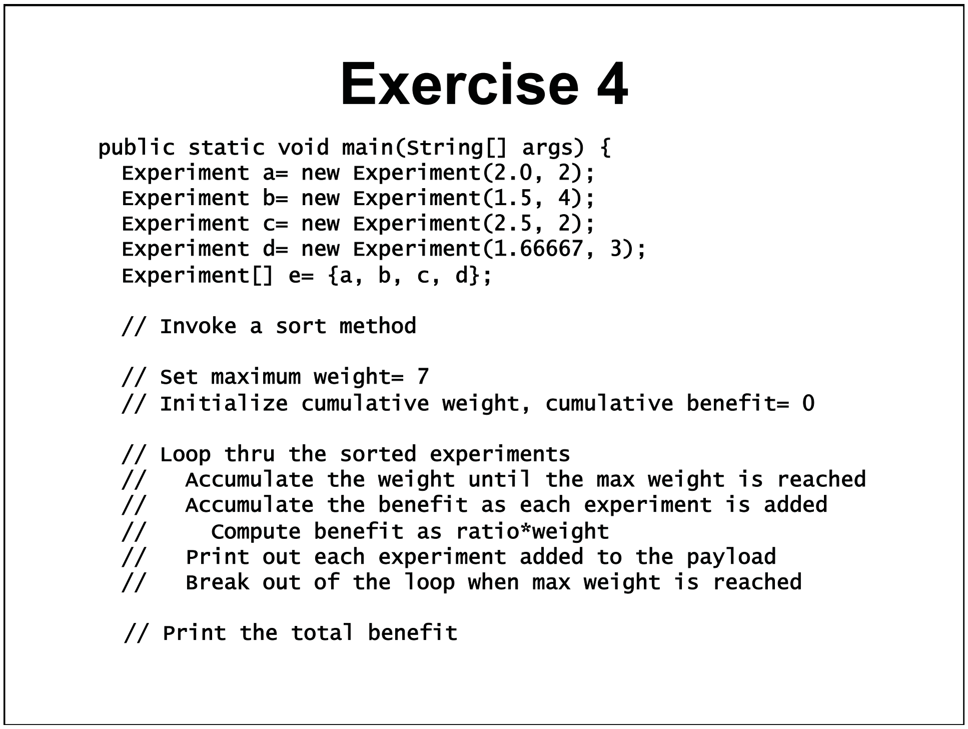
Exercise 3

```
import java.util.*;

public class Satellite {
    private static class Experiment implements
      Comparable<Experiment> {
         private double ratio;
         private int weight;

         public Experiment(double r, int w) {
             ratio= r;
             weight= w;
         }

        // Complete compareTo method. Sort in descending order.

         public String toString() {
            return (" Experiment: benefit: "+ (ratio*weight) +
                " weight: "+ weight+ " ratio: "+ ratio);
        }
    }
```



Exercise 4: Sorting, part 2

- In Satellite’s main():
  - We created the 4 Experiments from the earlier slide
  - We put the experiments in an array of Experiments
  - You should call a sort method to put the experiments in benefit/weight order
  - Set the weight limit= 7
  - Loop through the sorted experiments, and select the best ones to go on the satellite that fit within the weight limit
    - Keep track of cumulative weight, cumulative benefit
    - Be careful about floating point roundoff error!
  - Print out your solution:
    - Experiments to go on the satellite
    - Total weight
    - Total benefit



Exercise 4

```
public static void main(String[] args) {
  Experiment a= new Experiment(2.0, 2);
  Experiment b= new Experiment(1.5, 4);
  Experiment c= new Experiment(2.5, 2);
  Experiment d= new Experiment(1.66667, 3);
  Experiment[] e= {a, b, c, d};

  // Invoke a sort method

  // Set maximum weight= 7
  // Initialize cumulative weight, cumulative benefit= 0

  // Loop thru the sorted experiments
  //   Accumulate the weight until the max weight is reached
  //   Accumulate the benefit as each experiment is added
  //     Compute benefit as ratio*weight
  //   Print out each experiment added to the payload
  //   Break out of the loop when max weight is reached

  // Print the total benefit
```

1.00 / 1.001 / 1.002 Introduction to Computers and Engineering Problem Solving
Spring 2012