Lecture 24

Interfaces
(and More Arrays)

Announcements for This Lecture

Material

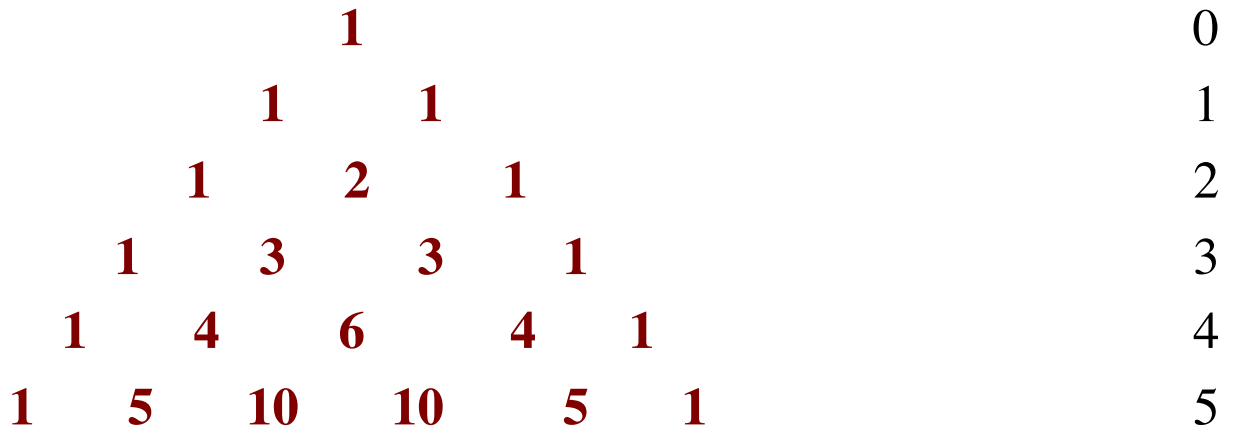
- Section 12.1
  - Compare with section 4.7
  - Relevant to assignment
- Next week is wrap up
  - **Tue**: Leaving DrJava
  - **Thu**: Where to from here?
- Review sessions in 2 weeks
  - Details next week

Assignments

- A6 still being graded
  - Done by Saturday
- Work on Assignment A7
  - Should have read by now
  - Keep track of the dates
    - Makes it manageable
    - Major push this weekend
  - Due Saturday after classes

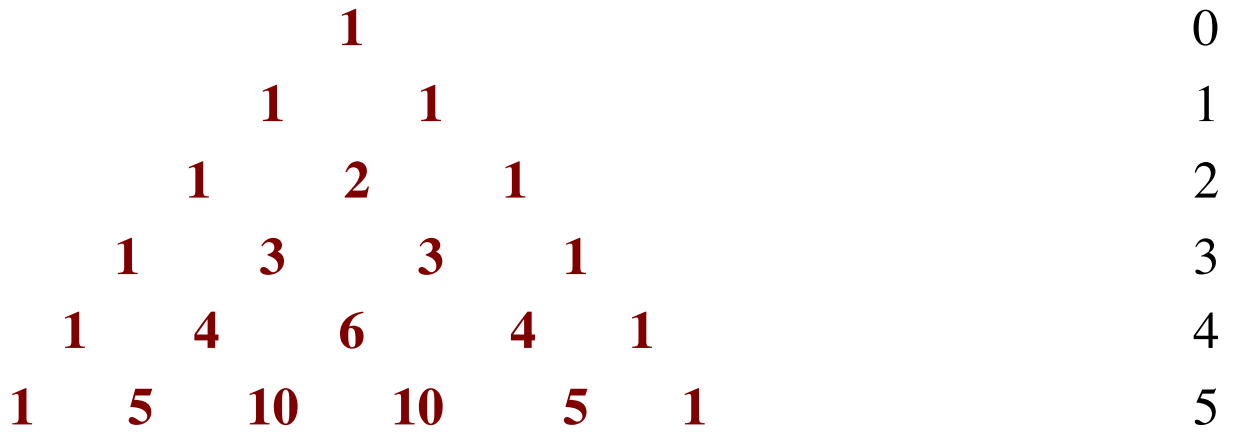
Carry over from last time…

Pascal's Triangle



- Creating the triangle:
  - The first and last entries on each row are 1.
  - Each other entry is the sum of the two entries above it
  - Row r has r+1 values.

Pascal's Triangle



- Entry p[i][j] = number of ways i elements can be chosen from a set of size j !
- p[i][j] = "i choose j" = $\binom{i}{j}$

Recursive formula:

for $0 < i < j$, $p[i][j] = p[i-1][j-1] + p[i-1][j]$

Pascal's Triangle

1	0
1 1	1
1 2 1	2
1 3 3 1	3
1 4 6 4 1	4
1 5 10 10 5 1	5
	…

- **Binomial Theorem**: Row r gives the coefficients of $(x + y)^r$
  - $(x + y)^2 = 1x^2 + 2xy + 1y^2$
  - $(x + y)^3 = 1x^3 + 3x^2y + 3xy^2 + 1y^3$
  - $(x + y)^r = \sum_{0 \le k \le r} (k \text{ choose } r)\, x^k y^{r-k}$

Ragged Arrays for Pascal's Triangle

```
/** Yields: ragged array of first n rows of Pascal's triangle. Precondition: 0 ≤ n */
public static int[][] pascalTriangle(int n) {
    int[][] b= new int[n][];     // First n rows of Pascal's triangle
    // invariant: rows 0..i-1 have been created
    for (int i = 0; i != b.length; i=  i+1) {
        b[i]= new int[i+1];     // Create row i of Pascal's triangle
        b[i][0]=  1;            // Calculate row i of Pascal's triangle
        // invariant b[i][0..j-1] have been created
        for (int j= 1; j < i; j= j+1) {
            b[i][j]=  b[i-1][j-1] + b[i-1][j];
        }
        b[i][i]= 1;
    }
    return b;
}
```

Summing Up a Multidimensional Array

```
/** Yields: Sum of elements of b.
  *   Precondition: b is an Integer or an array with base type Integer. */
public static int sum(Object b) {
      if (b instanceof Object[]) {
         Object[] bb= (Object[]) b;
         int sum= 0;
         //inv: sum = sum of b[0..k-1]
         for (int k= 0; k < bb.length; k= k+1) {
            sum= sum + sum(bb[k]);
         }
         return sum;
      }
      // { b has type Integer }
      return 0 + (Integer) b;
}
```

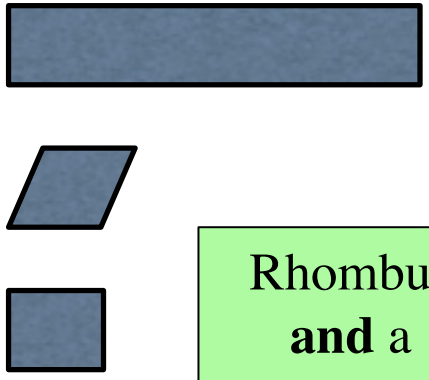
Recursive call on nested array

Base Case

New Topic: Interfaces

A Subclassing Example

- Classes for Shapes:
  - Rectangle: All angles equal
  - Rhombus: All sides same length
  - Square: All angles equal and all sides same length



- A square inherits from both rectangle and rhombus
  - `public class Rectangle { ... }`
  - `public class Rhombus { ... }`
  - `public class Square extends Rectangle, Rhombus { ... }`

Problem: Can Only Extend One Class

```
public class C extends C1, C2 { ... }
```

```
public class C1 {
   public int m() {
      return 2;
   }

   public int p() {
      return ...;
   }
}
```

```
public class C2 {
   public int m() {
      return 3;
   }

   public int q() {
      return ...;
   }
}
```

Which m() is inherited?

Problem: Can Only Extend One Class

```
public class C extends C1, C2 { ... }
```

```
public abstract class C1 {
   public abstract int m();
   public abstract int p();
}
```

```
public abstract class C2 {
   public abstract int m();
   public abstract int q();
}
```

- This is much better
  - Method bodies are not given
  - Nothing to inherit (or confuse)
- But still not allowed by Java

Java must have a guarantee that **all** the methods are abstract.

Use an Interface

```
public class C implements C1, C2 { ... }
```

```
public interface C1 {
   public int m();
   public int p();
}
```

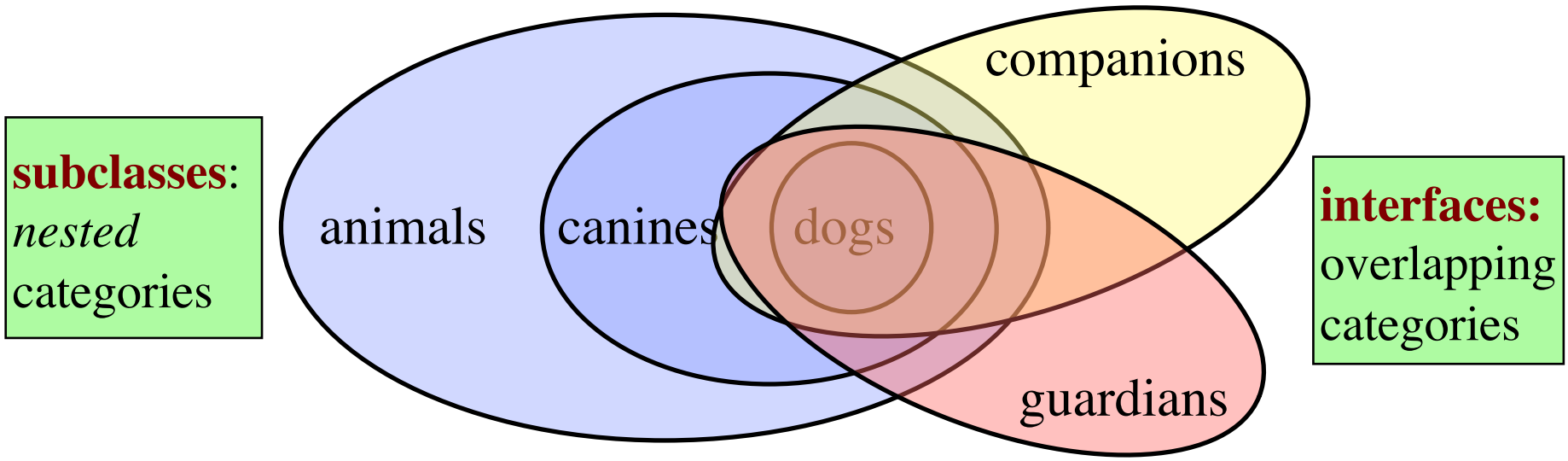
```
public interface C2 {
   public int m();
   public int q();
}
```

- All methods in an interface are abstract
  - No need for "abstract" keyword
  - Technically, "public" is also redundant (and is optional)
- **Example**: java.awt.event.ActionListener

Reading Class Definitions

```
public class Canine extends Animal { ... }
public class Dog extends Canine implements Companion, Guardian {...}
```

- Canines **are** animals. Dogs **are** canines.
- Dogs also **can serve as** companions or as guardians.



Application: Generalized Sorting

- Sorting is general, but notion of "<" may change
  - Recommender systems sort by quality, reviews, etc.
  - Travel sites sort by price, departure, etc.
  - Also, ascending vs. descending order
- Do not want to write many sort procedures:
  - `public void sort(int[] arr) {...}`
  - `public void sort(double[] arr) {...}`
  - `public void sort(Movie[] arr) {...}`
  - `public void sort(Flight[] arr) {...}`
- What if they all had a **comparison method**?

Interface java.util.Comparable

```
/** Comparable requires method compareTo*/
public interface Comparable {
   /** Yields: a negative integer if this object < c,
    *  Yields: 0 if this object = c,
    *  Yields: a positive integer if this object > c.
    *  Throws a ClassCastException if c cannot
    *  be cast to the class of this object. */
   int compareTo(Object c);
}
```

abstract method: body replaced by ;

Every class that *implements* Comparable must override compareTo(Object).

Implementing Classes

- Boolean
- Byte
- Double
- Integer
- …
- String
- Calendar
- Time
- Timestamp
- …

Using an Interface as a Type

```
/** Swap b[i] and b[j] to put larger in b[j]  */
public static void swap(Comparable [] b, int i, int j) {
    if (b[j].compareTo(b[i]) < 0) {
        Comparable temp= b[i];
        b[i]= b[j];
        b[j]= temp;
    }
}

public class Movie implements Comparable {
    String name;
    /** Yields -1, 0, or +1 if this Movie's name comes alphabetically before, at, or after c.
     *   Throws a ClassCastException if c cannot be cast to Movie.*/
    public int compareTo(Object c) {
        return this.name.compareTo(((Movie) c).name); // String implements Comparable
    }
}
```

Declaring Your Own Interfaces

```
/** comment */
public interface <interface-name> {
    /** method spec for function*/
    int compareTo(...);
    /** method spec for procedure */
    void doSomething(...);
    /** explanation of constant x*/
    int x= 7;
}
```

Use ";" instead of a body

Methods are implicitly **public**.
Can add modifier if you wish.

Every field is implicitly **public**, **static**, and **final**.
You can put these modifiers on them if you wish.

Class Can Implement Many Interfaces

```
/** comment */
public class C implements Inter1, Inter2, Inter3{
   ...
}
```

- Implements three interfaces: Inter1, Inter2, and Inter3
  - Must implement methods in **all of them**
- **Example**: Recommendation systems
  - Need to determine similarity (Similar interface)
  - Need to sort on this similarity (Comparable interface)