Odds and Ends...

• Static vs. Non-static
  – Class vs. Instance states and methods
• Private, public, protected, and default
  – Visibility to class, subclasses, and packages
  – p. 1136
• Common Mistakes
  – Class contains all functionality
  – Has-a vs. Is-a

Misuse of Inheritance

```java
public class Point3D extends Point {
    private int z;
    public Point3D(int x, int y, int z) {
        super(x, y);
    }
    public void getZ() {
        return z;
    }
    public class Point {
        public void equals(Object o) {
            if (o instanceof Point) {
                Point other = (Point) o;
                return (x==other.x && y==other.y);
            }
            return false;
        }
    }
    • Why is this bad?
```
Misuse of Inheritance

```java
public class Point3D extends Point {
    private int z;
    public void Point3D(int x, int y, int z) {
        super(x, y);
    }
    public void getZ() {
        return z;
    }
    public void equals(Object o) {
        if(o instanceof Point3D) {
            Point3D p = (Point3D) o;
            return(getX()==p.getX()&&getY()==p.getY()&&getZ()==p.getZ());
        }
        return false;
    }
}
```

Why is this bad?

```java
p.equals(p3d);
p3d.equals(p);
```

Coding with polymorphism

- A variable of type T can hold an object of any subclass of T.

```java
Employee ed = new Secretary();
```  
- You can call any methods from the Employee class on ed.

```java
System.out.println(ed.getSalary());  // $0000.0
System.out.println(ed.getVacationDays());  // pink
```

- When a method is called with ed, it behaves as a Secretary.

```java
System.out.println(ed.takeDictation());  // not allowed
```

Casting references

- A variable can only call that type's methods, not a subtype's.

```java
Employee ed = new Lawyer();
int hours = ed.getHours();  // ok; it's in Employee
ed.sue();  // compiler error
```  
- The compiler's reasoning is, variable ed could store any kind of employee, and not all kinds know how to sue.

- To use Lawyer methods on ed, we can type-cast it.

```java
Lawyer theRealEd = (Lawyer) ed;
theRealEd.sue();  // ok
((Lawyer) ed).sue();  // shorter version
```
More about casting

- The code crashes if you cast an object too far down the tree.

```java
Employee eric = new Secretary();
(Secretary) eric).takeDictation("Hi"); // ok
(Secretary) eric).fileLegalBriefs(); // exception
// (Secretary object doesn't know how to file briefs)
```

- You can cast only up and down the tree, not sideways.

```java
Lawyer linda = new Lawyer();
(Secretary) linda).takeDictation("Hi"); // error
```

- Casting doesn't actually change the object's behavior.

```java
(Employee) linda).getVacationForm(); // pink (Lawyer's)
```

Another exercise

- Assume that the following classes have been declared:

```java
public class Snow {
    public void method2() {
        System.out.println("Snow 2");
    }
    public void method3() {
        System.out.println("Snow 3");
    }
}
```

```java
public class Rain extends Snow {
    public void method1() {
        System.out.println("Rain 1");
    }
    public void method2() {
        System.out.println("Rain 2");
    }
}
```

```java
public class Sleet extends Snow {
    public void method2() {
        System.out.println("Sleet 2");
        method3();
    }
    public void method3() {
        System.out.println("Sleet 3");
    }
}
```

```java
public class Fog extends Sleet {
    public void method1() {
        System.out.println("Fog 1");
    }
    public void method3() {
        System.out.println("Fog 3");
    }
}
```

Exercise
Exercise

What happens when the following examples are executed?

- **Example 1:**
  ```java
  Snow var1 = new Sleet();
  var1.method2();
  ```

- **Example 2:**
  ```java
  Snow var2 = new Rain();
  var2.method1();
  ```

- **Example 3:**
  ```java
  Snow var3 = new Rain();
  ((Sleet) var3).method3();
  ```

Technique 1: diagram

- Diagram the classes from top (superclass) to bottom.

```
class Snow
  method1
  method2
  method3

class Rain
  method1
  method2
  method3

class Sleet
  method1
  method2
  method3

class Fog
  method1
  method2
  method3
```

Technique 2: table

<table>
<thead>
<tr>
<th>method</th>
<th>Snow</th>
<th>Rain</th>
<th>Sleet</th>
<th>Fog</th>
</tr>
</thead>
<tbody>
<tr>
<td>method1</td>
<td></td>
<td>Rain 1</td>
<td></td>
<td>Fog 1</td>
</tr>
<tr>
<td>method2</td>
<td>Snow 2</td>
<td>Rain 2</td>
<td>Sleet 2</td>
<td></td>
</tr>
<tr>
<td>method3</td>
<td>Snow 3</td>
<td>Snow 3</td>
<td>Sleet 3</td>
<td>Fog 3</td>
</tr>
</tbody>
</table>

*Italic - inherited behavior
Bold - dynamic method call*
Example 1

• Example:

```java
Snow var1 = new Sleet();
var1.method2();
```

• Output:

```java
Sleet 2
Snow 2
Sleet 3
```

Example 2

• Example:

```java
Snow var2 = new Rain();
var2.method1();
```

• Output:

```
None!
There is an error, because Snow does not have a method1.
```

Example 3

• Example:

```java
Snow var3 = new Rain();
(Sleet) var3).method2();
```

• Output:

```
None!
There is an error because a Rain is not a Sleet.
```
Interfaces

- **interface**: A list of methods that a class can implement.
  - Inheritance gives you an is-a relationship and code-sharing.
    - A Lawyer object can be treated as an Employee, and Lawyer inherits Employee's code.
  - Interfaces give you an is-a relationship **without** code sharing.
    - A Rectangle object can be treated as a Shape.

Declaring an interface

```java
public interface name {
    public type name (type name, .... type name);
    public type name (type name, .... type name);
    ...
}
```

Example:

```java
public interface Vehicle {
    public double speed();
    public void setDirection(int direction);
}
```

- **abstract method**: A header without an implementation.
  - The actual body is not specified, to allow/force different classes to implement the behavior in their own way.

Shape interface

```java
public interface Shape {
    public double area();
    public double perimeter();
}
```

- This interface describes the features common to all shapes.
  (Every shape has an area and perimeter.)
Implementing an interface

```java
public class name implements interface {
    ... 
}
```

- Example:
```java
public class Circle implements Shape {
    ... 
}
```

A class can declare that it implements an interface.

- This means the class must contain each of the abstract methods in that interface. (Otherwise, it will not compile.)

Complete Circle class

```java
// Represents circles.
public class Circle implements Shape {
    private double radius;
    // Constructs a new circle with the given radius.
    public Circle(double radius) {
        this.radius = radius;
    }
    // Returns the area of this circle.
    public double area() {
        return Math.PI * radius * radius;
    }
    // Returns the perimeter of this circle.
    public double perimeter() {
        return 2.0 * Math.PI * radius;
    }
}
```

Complete Rectangle class

```java
// Represents rectangles.
public class Rectangle implements Shape {
    private double width;
    private double height;
    // Constructs a new rectangle with the given dimensions.
    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
    // Returns the area of this rectangle.
    public double area() {
        return width * height;
    }
    // Returns the perimeter of this rectangle.
    public double perimeter() {
        return 2.0 * (width + height);
    }
}
```
Complete Triangle class

// Represents triangles.
public class Triangle implements Shape {
    private double a;
    private double b;
    private double c;

    // Constructs a new Triangle given side lengths.
    public Triangle(double a, double b, double c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    // Returns this triangle’s area using Heron’s formula.
    public double area() {
        double s = (a + b + c) / 2.0;
        return Math.sqrt(s * (s - a) * (s - b) * (s - c));
    }

    // Returns the perimeter of this triangle.
    public double perimeter() {
        return a + b + c;
    }
}

Interfaces + polymorphism

- Interfaces don’t benefit the class so much as the client.
  - Interface’s is-a relationship lets the client use polymorphism.

    public static void printInfo(Shape s) {
        System.out.println("The shape: " + s);
        System.out.println("area : " + s.area());
        System.out.println("perim: " + s.perimeter());
    }

- Any object that implements the interface may be passed.

    Circle circ = new Circle(12.0);
    Rectangle rect = new Rectangle(4, 7);
    Triangle tri = new Triangle(5, 12, 13);
    printInfo(circ);
    printInfo(tri);
    printInfo(rect);
    Shape[] shapes = {tri, circ, rect};

Interface diagram

- Arrow goes up from class to interface(s) it implements.
  - There is a supertype-subtype relationship here;
  - e.g., all Circles are Shapes, but not all Shapes are Circles.
  - This kind of picture is also called a UML class diagram.