Take a half-sheet of paper.

Please write down:
One good thing (going well in class)
   label it “good”
One bad thing (label it bad)
Or really bad thing (label it ugly)

Keep your paper for now.
You'll use it for an in-class exercise later.
Exception inheritance

- All exceptions extend from a common superclass `Exception`.

Exceptions and errors

- There are also `Errors`, which represent serious Java problems.
  - `Error` and `Exception` have common superclass `Throwable`.
  - You can catch an `Error` (but you probably shouldn't).

Java collection framework

- `Inheritance`: Forming new classes based on existing ones.
  - `superclass`: Parent class being extended.
  - `subclass`: Child class that inherits behavior from superclass.
    - `gets a copy of every field and method from superclass`.

Inheritance

- `inheritance`: Forming new classes based on existing ones.
  - A way to share/reuse code between two or more classes.
  - `superclass`: Parent class being extended.
  - `subclass`: Child class that inherits behavior from superclass.
    - Gets a copy of every field and method from superclass.
An Employee class

```java
public class Employee {
    ...
    public int getHours() {
        return 40; // works 40 hours / week
    }
    public double getSalary() {
        return 40000.0; // $40,000.00 / year
    }
    public int getVacationDays() {
        return 10; // 2 weeks' paid vacation
    }
    public String getVacationForm() {
        return "yellow"; // use the yellow form
    } // ...
}
```

- Lawyers, Secretaries, etc. have similar behavior to the above.
- How to implement those classes without redundancy?

Inheritance syntax

```java
public class name extends superclass {
    ...
}
```

- Example:
```java
public class Lawyer extends Employee {
    ...
}
```

- By extending Employee, each Lawyer object now:
  - receives a copy of each method from Employee automatically
  - can be treated as an Employee by client code

Overriding methods

- **Override**: To replace a superclass's method by writing a new version of that method in a subclass.

  - No special syntax is required to override a method. Just write a new version of it in the subclass.

```java
public class Lawyer extends Employee {
    // overrides getSalary method in Employee class;
    // give Lawyers a $5K raise
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.00;
    }
}
```

- This version makes sure that Lawyers always make $5K more than Employees, even if the Employee's salary changes.

super keyword

- Subclasses can call overridden methods with `super.method(parameters)`

  - Example:
```java
public class Lawyer extends Employee {
    // give Lawyers a $5K raise (better)
    public double getSalary() {
        double baseSalary = super.getSalary();
        return baseSalary + 5000.00;
    }
}
```

- This version makes sure that Lawyers always make $5K more than Employees, even if the Employee's salary changes.
**Levels of inheritance**

- Multiple levels of inheritance in a hierarchy are allowed.
  - Example: A legal secretary is the same as a regular secretary but makes more money ($45,000) and can file legal briefs.

  ```java
  public class LegalSecretary extends Secretary {
      ...
  }
  ``

  - Exercise: Complete the `LegalSecretary` class.

**LegalSecretary class**

```java
// A class to represent legal secretaries.
public class LegalSecretary extends Secretary {
    public void fileLegalBriefs() {
        System.out.println("I could file all day!");
    }
    public double getSalary() {
        return 45000.0;  // $45,000.00 / year
    }
}
```

**Responding to Feedback**

**Problem:** Assignments are too confusing for some students  
**Solution:** More time in lab will be devoted to assignments

**Problem:** Pace of the course is too slow for some students  
**Solution:** We'll move slightly faster, and I'll try to allude to more advanced topics more often

**Problem:** Getting help by email is unreliable  
**Solution:** Still thinking about this one.

**Hints for Getting Help**

- Be specific. Don't just say, “my code doesn't work.”
- Describe the debugging steps you've already taken.
  - Did you read the book or look up the API reference?
  - Did you try to come up with a simpler test case?
  - If asking by email, allow 24 hours.
Exercise: StutterIntList

- Implement StutterIntList, an ArrayIntList that adds several copies of an integer whenever you call the add method.
- The constructor for StutterIntList should accept the stretchFactor or number of repetitions as an argument. The initial capacity should be a second, optional argument.
- Override both add(value) and add(value, index)

Example:
StutterIntList l = new StutterIntList(5);
l.add(1);
l.add(2);
l.add(3, 2);
System.out.println(l);
Output:
[1, 1, 3, 3, 3, 3, 1, 1, 2, 2, 2, 2, 2]

Exercise solution

```java
public class StutterIntList extends ArrayIntList {
    private int stretch;

    public StutterIntList(int stretchFactor) {
        super();
        stretch = stretchFactor;
    }

    public StutterIntList(int stretchFactor, int capacity) {
        super(capacity);
        stretch = stretchFactor;
    }

    public void add(int value) {
        for (int i = 1; i <= stretch; i++) {
            super.add(value);
        }
    }

    public void add(int index, int value) {
        for (int i = 1; i <= stretch; i++) {
            super.add(index, value);
        }
    }

    public int getStretch() {
        return stretch;
    }
}
```

Subclasses and fields

```java
public class Employee {
    private double salary;
    ...
}

public class Lawyer extends Employee {
    ...
    public void giveRaise(double amount) {
        salary += amount; // error; salary is private
    }
}
```

- Inherited private fields/methods cannot be directly accessed by subclasses. (The subclass has the field, but it can't touch it.)
- How can we allow a subclass to access/modify these fields?
Protected fields/methods

```java
protected type name; // field

protected type name(type name, ..., type name) {
    statement(s); // method
}
```

- A protected field or method can be seen/called only by:
  - the class itself, and its subclasses
  - also by other classes in the same “package”
  - useful for allowing selective access to inner class implementation

```java
public class Employee {
    protected double salary;
    ...
}
```

Object variables

- You can store any object in a variable of type Object.
  ```java
  Object o1 = new Point(5, -3);
  Object o2 = "hello there";
  Object o3 = new Scanner(System.in);
  ```

- An Object variable only knows how to do general things.
  ```java
  String s = o1.toString(); // ok
  int len = o2.length(); // error
  String line = o3.nextLine(); // error
  ```

- You can write methods that accept an Object parameter.
  ```java
  public void checkForNull(Object o) {
      if (o == null) {
          throw new IllegalArgumentException();
      }
  }
  ```

Class Object

- All types of objects have a superclass named Object.
  - Every class implicitly extends Object

- The Object class defines several methods:
  - public String toString()
    Returns a text representation of the object, often so that it can be printed.
  - public boolean equals(Object other)
    Compare the object to any other for equality. Returns true if the objects have equal state.

Polymorphism
Coding with polymorphism

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

  \[ \text{Employee ed} = \text{new Lawyer();} \]

- You can call any methods from the \text{Employee} class on \( ed \).

- When a method is called on \( ed \), it behaves as a \text{Lawyer}.

  \[
  \text{System.out.println(} \text{ed.getSalary();} \gg 50000.0 \\
  \text{System.out.println(} \text{ed.getVacationForm();} \gg \text{pink}\]

Polymorphism and parameters

- You can pass any subtype of a parameter's type.

  ```java
  public class EmployeeMain {
    public static void main(String[] args) {
      Lawyer lisa = new Lawyer();
      Secretary steve = new Secretary();
      printInfo(lisa);
      printInfo(steve);
    }
  }
  
  public static void printInfo(Employee empl) {
    System.out.println("salary: "+ empl.getSalary();
    System.out.println("v.days: "+ empl.getVacationDays());
    System.out.println("v.form: "+ empl.getVacationForm());
  }
  ```

  Output:
  
<table>
<thead>
<tr>
<th>salary</th>
<th>v.days</th>
<th>v.form</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000.0</td>
<td>15</td>
<td>pink</td>
</tr>
<tr>
<td>50000.0</td>
<td>10</td>
<td>yellow</td>
</tr>
</tbody>
</table>

Polymorphism and arrays

- Arrays of superclass types can store any subtype as elements.

  ```java
  public class EmployeeMain2 {
    public static void main(String[] args) {
      Employee[] e = { new Lawyer(),
                      new Secretary(),
                      new Marketer(),
                      new LegalSecretary() ];
      for (int i = 0; i < e.length; i++) {
        System.out.println("salary: "+ e[i].getSalary();
        System.out.println("v.days: "+ e[i].getVacationDays());
        System.out.println();
      }
    }
  }
  ```

  Output:
  
<table>
<thead>
<tr>
<th>salary</th>
<th>v.days</th>
<th>v.form</th>
</tr>
</thead>
<tbody>
<tr>
<td>50000.0</td>
<td>15</td>
<td>pink</td>
</tr>
<tr>
<td>50000.0</td>
<td>10</td>
<td>yellow</td>
</tr>
<tr>
<td>60000.0</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>55000.0</td>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Polymorphism problems

- 4-5 classes with inheritance relationships are shown.

- A client program calls methods on objects of each class.

- You must read the code and determine the client's output.

- We always put such a question on our final exams!
A polymorphism problem

```java
public class Foo {
    public void method1() { System.out.println("A1"); }
    public void method2() { System.out.println("A2"); }
    public String toString() { return "foo"; }
}
public class Bar extends Foo {
    public void method2() { System.out.println("B2"); }
}
public class Baz extends Foo {
    public void method1() { System.out.println("C1"); }
    public String toString() { return "baz"; }
}
public class Mumble extends Baz {
    public void method2() { System.out.println("D2"); }
}
```

• What would be the output of the following client code?

```java
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

Diagramming the classes

- Add classes from top (superclass) to bottom (subclass).
- Include all inherited methods.

Finding output with tables

<table>
<thead>
<tr>
<th>method</th>
<th>Foo</th>
<th>Bar</th>
<th>Baz</th>
<th>Mumble</th>
</tr>
</thead>
<tbody>
<tr>
<td>method1</td>
<td>foo 1</td>
<td>foo 1</td>
<td>baz 1</td>
<td>baz 1</td>
</tr>
<tr>
<td>method2</td>
<td>foo 2</td>
<td>bar 2</td>
<td>foo 2</td>
<td>mumble 2</td>
</tr>
<tr>
<td>toString</td>
<td>foo</td>
<td>baz</td>
<td>baz</td>
<td></td>
</tr>
</tbody>
</table>
Polymorphism answer

```java
Foo[] pity = {new Baz(), new Bar(), new Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

- **Output:**
  - baz
  - baz 1
  - foo
  - foo 1
  - bar 2
  - baz
  - baz 1
  - mumble 2
  - foo
  - foo 1
  - foo 2

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
((LegalSecretary) 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.
  - It just gets the code to compile/run.

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

Run-Time Type Information

- You can check the legality of a cast before you do it:

```java
Lawyer linda = new Lawyer();
if (linda instanceof Secretary) {
    ((Secretary) linda).takeDictation("hi");
}
```

- It’s generally best to avoid casts as much as possible.
- In many cases, reliance on `instanceof` can be replaced by proper use of polymorphism.
How Does Inheritance Work?

- When class B extends class A, the fields in class A are a subset:

  (Employee) ed

  Secretary class

  - Double Salary
  - String VacationForm
  - Int VacationDays
  - Double TypingSpeed

  Employee class

- Therefore, every Secretary can be treated as an Employee by only looking at the first three fields

How Does Polymorphism Work?

- A subclass has all the methods of the superclass:

  Method Table / vtable

  Special Class Data

  - String toString()
  - Double getSalary()
  - Int getVacationDays()
  - Double getTypingSpeed()
  - String getStapler()

- Overriding a method changes the entry in the method table for this class.
- When we cast a Secretary as an Employee, the method table is unchanged: toString() and getSalary() still point to the Secretary-specific code

Polymorphism vs. Overloading

- Overloading: Two or more methods with different parameters and the same name. Which method to call is chosen statically (at compile time).

  public void add(int value);
  public void add(int value, int index);
  public void add(ArrayIntList list);

- Polymorphism: Related classes define a method with the same name and parameters. Which method to call is chosen dynamically (at runtime).

Real Interview Question

```java
/* What does the following program print? */
public class Test {
    public boolean equals( Test other ) {
        System.out.println( "Inside of Test.equals" );
        return false;
    }
    public static void main( String [] args ) {
        public void add(int value);
        public void add(int value, int index);
        public void add(ArrayIntList list);

        public static void main( String [] args ) {
            Object t1 = new Test();
            Object t2 = new Test();
            Object t3 = new Test();
            Object t4 = new Object();
            System.out.println("1");
            t1.equals(t2);
            System.out.println("2");
            t1.equals(t3);
            System.out.println("3");
            t1.equals(t1);
            System.out.println("4");
            t3.equals(t3);
            System.out.println("5");
            t3.equals(t2);
        }
    }
}```
**Interfaces**

**Relatedness of types**

Write a set of Circle, Rectangle, and Triangle classes.

- Certain operations that are common to all shapes.
  - perimeter - distance around the outside of the shape
  - area - amount of 2D space occupied by the shape
- Every shape has them but computes them differently.

**Shape area, perimeter**

- Rectangle (as defined by width \( w \) and height \( h \)):
  - area = \( w \times h \)
  - perimeter = \( 2w + 2h \)

- Circle (as defined by radius \( r \)):
  - area = \( \pi r^2 \)
  - perimeter = \( 2 \pi r \)

- Triangle (as defined by side lengths \( a, b, \) and \( c \)):
  - area = \( \sqrt{s(s-a)(s-b)(s-c)} \)
    where \( s = \frac{1}{2} (a + b + c) \)
  - perimeter = \( a + b + c \)

**Common behavior**

- Write shape classes with methods `perimeter` and `area`.
- We’d like to be able to write client code that treats different kinds of shape objects in the same way, such as:
  - Write a method that prints any shape’s area and perimeter.
  - Create an array of shapes that could hold a mixture of the various shape objects.
  - Write a method that could return a rectangle, a circle, a triangle, or any other shape we’ve written.
  - Make a DrawingPanel display many shapes on screen.
**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*.
  - Analogous to the idea of roles or certifications:
    - "I'm certified as a CPA accountant. That means I know how to compute taxes, perform audits, and do consulting."
    - "I'm certified as a *Shape*. That means I know how to compute my area and perimeter."

**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 its 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 Bicycle implements Vehicle {
      ...
  }
  ```

- 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.)

  (What must be true about the *Bicycle* class for it to compile?)
Interface requirements

• If a class claims to be a Shape but doesn't implement the area and perimeter methods, it will not compile.

  – Example:
  public class Banana implements Shape {
      ...
  }

  – The compiler error message:
    Banana.java:1: Banana is not abstract and does not override abstract method area() in Shape
    public class Banana implements Shape {
        ^

Complete Circle class

// 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

// 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};
```

Abstract Classes

Specify required functionality without implementation (like an interface) but provide partial implementation (like a superclass):

```
public abstract class Foo {
    private int bar;
    public Foo(int bar) { this.bar = bar; }
    public String toString() { return "foo"; }
    // Do something important.
    public abstract void baz();
}
```

An abstract class may have a constructor (or several), but it can never be instantiated using new, since its functionality is incomplete. (Why would you want a constructor then?)

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.

Java collections framework
Comparable (10.2)

public interface Comparable<E> {
    public int compareTo(E other);
}

• A class can implement the Comparable interface to define a natural ordering function for its objects.

• A call to your compareTo method should return:
  a value < 0 if the other object comes "before" this one,
  a value > 0 if the other object comes "after" this one,
  or       0 if the other object is considered "equal" to this.

• If you want multiple orderings, use a Comparator instead (see Ch. 13.1)

compareTo and collections

• You can use an array or list of strings with Java’s included binary search method because it calls compareTo internally.
  String[] a = {"al", "bob", "cari", "dan", "mike"};
  int index = Arrays.binarySearch(a, "dan"); // 3

• Java’s TreeSet/Map use compareTo internally for ordering.
  Set<String> set = new TreeSet<String>();
  for (String s : a) {
    set.add(s);
  }
  System.out.println(s);
  // [al, bob, cari, dan, mike]

Using compareTo

• compareTo can be used as a test in an if statement.
  String a = "alice";
  String b = "bob";
  if (a.compareTo(b) < 0) { // true
    ...
  }

<table>
<thead>
<tr>
<th>Primitives</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (a &lt; b)</td>
<td>if (a.compareTo(b) &lt; 0)</td>
</tr>
<tr>
<td>if (a &lt;= b)</td>
<td>if (a.compareTo(b) &lt;= 0)</td>
</tr>
<tr>
<td>if (a == b)</td>
<td>if (a.compareTo(b) == 0)</td>
</tr>
<tr>
<td>if (a != b)</td>
<td>if (a.compareTo(b) != 0)</td>
</tr>
<tr>
<td>if (a &gt; b)</td>
<td>if (a.compareTo(b) &gt; 0)</td>
</tr>
</tbody>
</table>

Ordering our own types

• We cannot binary search or make a TreeSet/Map of arbitrary types, because Java doesn’t know how to order the elements.
  – The program compiles but crashes when we run it.

  Set<HtmlTag> tags = new TreeSet<HtmlTag>();
  tags.add(new HtmlTag("body", true));
  tags.add(new HtmlTag("b", false));
  ...

  Exception in thread "main" java.lang.ClassCastException
  at java.util.TreeSet.add(TreeSet.java:238)
Comparable template

```java
public class name implements Comparable<name> { 
    ...
    public int compareTo(name other) {
        ...
    }
}
```

Comparable example

```java
public class Point implements Comparable<Point> {
    private int x;
    private int y;
    ...
    // sort by x and break ties by y
    public int compareTo(Point other) {
        if (x < other.x) {
            return -1;
        } else if (x > other.x) {
            return 1;
        } else if (y < other.y) {
            return -1;  // same x, smaller y
        } else if (y > other.y) {
            return 1;   // same x, larger y
        } else {
            return 0;   // same x and same y
        }
    }
}
```

compareTo tricks

- **subtraction trick** - Subtracting related numeric values produces the right result for what you want `compareTo` to return:

  ```java
  // sort by x and break ties by y
  public int compareTo(Point other) {
      if (x != other.x) {
          return x - other.x;  // different x
      } else {
          return y - other.y;  // same x; compare y
      }
  }
  ```

  - The idea:
    - if `x > other.x`, then `x - other.x > 0`
    - if `x < other.x`, then `x - other.x < 0`
    - if `x == other.x`, then `x - other.x == 0`

  - NOTE: This trick doesn’t work for doubles (but see Math.signum)

compareTo tricks 2

- **delegation trick** - If your object’s fields are comparable (such as strings), use their `compareTo` results to help you:

  ```java
  // sort by employee name, e.g. "Jim" < "Susan"
  public int compareTo(Employee other) {
      return name.compareTo(other.getName());
  }
  ```

- **toString trick** - If your object’s `toString` representation is related to the ordering, use that to help you:

  ```java
  // sort by date, e.g. "09/19" > "04/01"
  public int compareTo(Date other) {
      return toString().compareTo(other.toString());
  }
  ```