Building Java Programs

Chapter 4+5: Inheritance and Interfaces

These are the slides of the textbook, modified/corrected where necessary. The chapters have been re-enumerated according to CMPE160.

Chapter outline

- background
  - categories of employees
  - relationships and hierarchies
- inheritance programming
  - creating subclasses
  - overriding behavior
  - multiple levels of inheritance
  - interacting with the superclass using the `super` keyword
  - inheritance and design
- polymorphism
  - "polymorphism mystery" problems
- interfaces

The software crisis

- **software engineering**: The practice of conceptualizing, designing, developing, documenting, and testing large-scale computer programs.

- Large-scale projects face many issues:
  - getting many programmers to work together
  - getting code finished on time
  - avoiding redundant code
  - finding and fixing bugs
  - maintaining, improving, and reusing existing code

- **code reuse**: The practice of writing program code once and using it in many contexts.
Employee analogy

Consider a law firm with many types of employees.

- common rules: hours, vacation time, benefits, regulations, ...
  - all employees attend common orientation to learn general rules
  - each employee receives 20-page manual of the common rules

- each subdivision also has specific rules
  - employee attends a subdivision-specific orientation to learn them
  - employee receives a smaller (1-3 page) manual of these rules
  - smaller manual adds some rules and also changes some rules from the large manual ("use the pink form instead of yellow form" ...)

Separating behavior

- Why not just have a 22 page Lawyer manual, a 21-page Secretary manual, a 23-page Marketer manual, etc.?

- Some advantages of the separate manuals:
  - maintenance: If a common rule changes, we'll need to update only the common manual.
  - locality: A person can look at the lawyer manual and quickly discover all rules that are specific to lawyers.

- Some key ideas from this example:
  - It's useful to be able to describe general rules that will apply to many groups (the 20-page manual).
  - It's also useful for a group to specify a smaller set of rules for itself, including being able to replace rules from the overall set.

Is-a relationships, hierarchies

- is-a relationship: A hierarchical connection where one category can be treated as a specialized version of another.
  - every marketer is an employee
  - every legal secretary is a secretary

- inheritance hierarchy: A set of classes connected by is-a relationships that can share common code.
  - Often drawn as a downward tree of connected boxes or ovals representing classes:

Employee regulations

- Consider the following employee regulations:
  - Employees work 40 hours per week.
  - Employees make $40,000 per year, except legal secretaries who make $5,000 extra per year ($45,000 total), and marketers who make $10,000 extra per year ($50,000 total).
  - Employees have 2 weeks of paid vacation leave per year, except lawyers who get an extra week (a total of 3).
  - Employees should use a yellow form to apply for leave, except for lawyers who use a pink form.

- Each type of employee has some unique behavior:
  - Lawyers know how to sue.
  - Marketers know how to advertise.
  - Secretaries know how to take dictation.
  - Legal secretaries know how to prepare legal documents.
General employee code

// A class to represent employees in general (20-page manual).
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
    }
}

Exercise: Implement class Secretary, based on the previous employee regulations.

Redundant secretary code

// A redundant class to represent secretaries.
public class Secretary {
    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
    }
    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}

Desire for code-sharing

- The takeDictation method is the only unique behavior in the Secretary class.
- We'd like to be able to say the following:

// A class to represent secretaries.
public class Secretary {
    //copy all the contents from Employee class.
    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}

Inheritance

- inheritance: A way to form new classes based on existing classes, taking on their attributes/behavior.
  - a way to group related classes
  - a way to share code between two or more classes
- We say that one class can extend another by absorbing its state and behavior.
  - superclass: The parent class that is being extended.
  - subclass: The child class that extends the superclass and inherits its behavior.
    - The subclass receives a copy of every field and method from its superclass.
Inheritance syntax

- Creating a subclass, general syntax:
  public class <name> extends <superclass name> {

  
  Example:
  public class Secretary extends Employee {
      ....
  }

- By extending Employee, each Secretary object now:
  - receives a getHours, getSalary, getVacationDays, and getVacationForm method automatically
  - can be treated as an Employee by any other code (seen later)
    (e.g. a Secretary could be stored in a variable of type Employee or stored as an element of an Employee[])

Improved secretary code

```java
// A class to represent secretaries.
public class Secretary extends Employee {
    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: "+text);
    }
}
```

- Now we only have to write the portions that are unique to each type.
  - Secretary inherits getHours, getSalary, getVacationDays, and getVacationForm methods from Employee.
  - Secretary adds the takeDictation method.

Implementing Lawyer

- Let's implement a Lawyer class.

- Consider the following employee regulations:
  - Lawyers get an extra week of paid vacation (a total of 3).
  - Lawyers use a pink form when applying for vacation leave.
  - Lawyers have some unique behavior: they know how to sue.

- The problem: We want lawyers to inherit most of the behavior of the general employee, but we want to replace certain parts with new behavior.

Overriding methods

- **override**: To write a new version of a method in a subclass that replaces the superclass' version.

  There is no special syntax for overriding.
  To override a superclass method, just write a new version of it in the subclass. This will replace the inherited version.

- Example:
  ```java
  public class Lawyer extends Employee {
      // overrides getVacationForm method in Employee class
      public String getVacationForm() {
          return "pink";
      }
      ...
  }
  ```

- Exercise: Complete the Lawyer class.
Complete Lawyer class

// A class to represent lawyers.
public class Lawyer extends Employee {
    // overrides getVacationForm from Employee class
    public String getVacationForm() {
        return "pink";
    }

    // overrides getVacationDays from Employee class
    public int getVacation() {
        return 15;    // 3 weeks vacation
    }

    public void sue() {
        System.out.println("I'll see you in court!");
    }
}

• Exercise: Now complete the Marketer class. Marketers make $10,000 extra ($50,000 total) and know how to advertise.

Levels of inheritance

• Deep hierarchies can be created by multiple levels of subclassing.
  • Example: The legal secretary is the same as a regular secretary except for making more money ($45,000) and being able to file legal briefs.

  public class LegalSecretary extends Secretary {
      ...
  }

  • Exercise: Complete the LegalSecretary class.

Complete LegalSecretary class

// 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() {    // $45,000.00 / year
        return 45000.0;
    }
}

Complete Marketer class

// A class to represent marketers.
public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }

    public double getSalary() {
        return 50000.0;    // $50,000.00 / year
    }
}
Interacting with the superclass: the super keyword

Changes to common behavior
- Imagine that a company-wide change occurs that affects all employees.
  - Example: Because of inflation, everyone is given a $10,000 raise.
  - The base employee salary is now $50,000.
  - Legal secretaries now make $55,000.
  - Marketers now make $60,000.
- We must modify our code to reflect this policy change.

Modifying the superclass
- This modified Employee class handles the new raise:

```
// A class to represent employees in general (20-page manual).
public class Employee {
    public int getHours() {
        return 40;  // works 40 hours / week
    }
    public double getSalary() {
        return 50000.0;  // $50,000.00 / year
    }
    ...
}
```
- What problem now exists in the code?
- The Employee subclasses are now incorrect.
  - They have overridden the getSalary method to return other values such as 45,000 and 50,000 that need to be changed.

An unsatisfactory solution
```
public class LegalSecretary extends Secretary {
    public double getSalary() {
        return 55000.0;
    }
    ...
}
```
```
public class Marketer extends Employee {
    public double getSalary() {
        return 60000.0;
    }
    ...
}
```
- The employee subtypes' salaries are tied to the overall base employee salary, but the subclasses' getSalary code does not reflect this relationship.
Calling overridden methods

- A subclass can call an overridden method with the `super` keyword.

- Calling an overridden method, syntax:
  
  ```java
  super . <method name> ( <parameter(s)> )
  ```

- Example:
  
  ```java
  public class LegalSecretary extends Secretary {
      public double getSalary() {
          double baseSalary = super.getSalary();
          return baseSalary + 5000.0;
      }
  }
  ```

- Exercise: Modify the `Lawyer` and `Marketer` classes to also use the `super` keyword.

Inheritance and constructors

- Imagine that we want to give employees more vacation days the longer they’ve been with the company.
  
  - For each year worked, we’ll award 2 additional vacation days.

- When an Employee object is constructed, we’ll pass in the number of years the person has been with the company.

- This will require us to modify our `Employee` class and add some new state and behavior.

- Exercise: Make the necessary modifications to the `Employee` class.

Improved subclasses

```java
public class Lawyer extends Employee {
    public String getVacationForm() {
        return "pink";
    }
    public int getVacationDays() {
        return super.getVacationDays() + 5;
    }
    public void sue() {
        System.out.println("I'll see you in court!");
    }
}
```

```java
public class Marketer extends Employee {
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }
    public double getSalary() {
        return super.getSalary() + 10000.0;
    }
}
```

Modified Employee class

```java
public class Employee {
    private int years;
    public Employee(int years) {
        this.years = years;
    }
    public int getHours() {
        return 40;
    }
    public double getSalary() {
        return 50000.0;
    }
    public int getVacationDays() {
        return 10 + 2 * years;
    }
    public String getVacationForm() {
        return "yellow";
    }
}
```
Problem with constructors

- Now that we've added the constructor to the Employee class, our subclasses do not compile. The error:

```java
Lawyer.java:2: cannot find symbol
symbol : constructor Employee()
location: class Employee
public class Lawyer extends Employee {
  ^
```

- The short explanation: Once we write a constructor (that requires parameters) in the superclass, we must now write constructors for our employee subclasses as well.

- The long explanation: (next slide)

The detailed explanation

- Constructors aren't inherited.
  - The Employee subclasses don't inherit the public Employee(int years) constructor.
  - Since our subclasses don't have constructors, they receive a default parameterless constructor that contains the following:
    ```java
    public Lawyer() {
        super(); // calls public Employee() constructor
    }
    ```
  - But our public Employee(int years) replaces the default Employee constructor.
  - Therefore all the subclasses' default constructors are now trying to call a non-existent default superclass constructor.

Calling superclass constructor

- Syntax for calling superclass's constructor:
  ```java
  super( <parameter(s)> );
  ```
- Example:
  ```java
  public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years); // call Employee constructor
    }
  }
  ```
- The call to the superclass constructor must be the first statement in the subclass constructor.
- Exercise: Make a similar modification to the Marketer class.

Modified Marketer class

```java
// A class to represent marketers.
public class Marketer extends Employee {
    public Marketer(int years) {
        super(years);
    }
    public void advertise() {
        System.out.println("Act now while supplies last!");
    }
    public double getSalary() {
        return super.getSalary() + 10000.0;
    }
}
```
- Exercise: Modify the Secretary subclass to make it compile:
  - Secretaries' years of employment are not tracked and they do not earn extra vacation for them.
  - Secretary objects are also constructed without a years parameter.
Modified Secretary class

// A class to represent secretaries.
public class Secretary extends Employee {
    public Secretary() {
        super(0);
    }

    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: "+text);
    }
}

Note that since the Secretary doesn't require any parameters to its constructor, the LegalSecretary now compiles without a constructor (its default constructor calls the parameterless Secretary constructor).

This isn't the best solution; it isn't that Secretaries work for 0 years, it's that they don't receive a bonus. How can we fix it?

Inheritance and fields

- Suppose that we want to give lawyers a $5000 raise for each year they've been with the company.
- The following modification doesn't work:

public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);
        
        public double getSalary() {
            return super.getSalary() + 5000 * years;
        }
    }

The error is the following:

Lawyer.java:7: years has private access in Employee
    return super.getSalary() + 5000 * years;
        ^

Private fields cannot be directly accessed from other classes, not even subclasses.
- One reason for this is to prevent malicious programmers from using subclassing to circumvent encapsulation.
- How can we get around this limitation?

Private access limitations

public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);
    }

    public double getSalary() {
        return super.getSalary() + 5000 * years;
    }
}

The error is the following:

Lawyer.java:7: years has private access in Employee
    return super.getSalary() + 5000 * years;
        ^

Private fields cannot be directly accessed from other classes, not even subclasses.
- One reason for this is to prevent malicious programmers from using subclassing to circumvent encapsulation.
- How can we get around this limitation?

Improved Employee code

Add an accessor for any field needed by the superclass.

public class Employee {
    private int years;

    public Employee(int years) {
        this.years = years;
    }

    public int getYears() {
        return years;
    }
}

public class Lawyer extends Employee {
    public Lawyer(int years) {
        super(years);

        public double getSalary() {
            return super.getSalary() + 5000 * getYears();
        }
    }

    public int getYears() {
        return years;
    }
}

Copyright 2006 by Pearson Education

Revisiting Secretary

- The Secretary class currently has a poor solution.
  - We set all Secretaries to 0 years because they do not get a vacation bonus for their service.
  - If we call `getYears` on a Secretary object, we'll always get 0.
  - This isn't a good solution; what if we wanted to give some other reward to all employees based on years of service?

- Let's redesign our Employee class a bit to allow for a better solution.

Improved Employee code

Let's separate the standard 10 vacation days from those that are awarded based on seniority.

```java
public class Employee {
    private int years;
    public Employee(int years) {
        this.years = years;
    }
    public int getVacationDays() {
        return 10 + getSeniorityBonus();
    }
    public int getSeniorityBonus() {
        return 2 * years;
    }
    // vacation days given for each year in the company
    public int getSeniorityBonus() {
        return 2 * years;
    }
    ...
}
```

- How does this help us improve the Secretary?

Improved Secretary code

The Secretary can selectively override the `getSeniorityBonus` method, so that when it runs its `getVacationDays` method, it will use this new version as part of the computation.

- Choosing a method at runtime like this is called dynamic binding.

```java
public class Secretary extends Employee {
    public Secretary(int years) {
        super(years);
    }
    // Secretaries don't get a bonus for their years of service.
    public int getSeniorityBonus() {
        return 0;
    }
    public void takeDictation(String text) {
        System.out.println("Taking dictation of text: " + text);
    }
}
```

Polymorphism

reading: 9.2
Polymorphism

- **polymorphism**: The ability for the same code to be used with several different types of objects and behave differently depending on the type of object used.

- A reference variable of a type T can legally refer to an object of any subclass of T.

```java
Employee person = new Lawyer(3);
System.out.println(person.getSalary()); // 65000.0
System.out.println(person.getVacationForm()); // pink
```

- You can call any methods from Employee on the variable person, but not any methods specific to Lawyer (such as sue).

- Once a method is called on the object, it behaves in its normal way (as a Lawyer, not as a normal Employee).

Polymorphism + parameters

- You can declare methods to accept superclass types as parameters, then pass a parameter of any subtype.

```java
public class EmployeeMain {
    public static void main(String[] args) {
        Lawyer lisa = new Lawyer(3);
        Secretary steve = new Secretary(2);
        printInfo(lisa);
        printInfo(steve);
    }
    public static void printInfo(Employee empl) {
        System.out.println("salary = " + empl.getSalary());
        System.out.println("days = " + empl.getVacationDays());
        System.out.println("form = " + empl.getVacationForm());
    }
}
```

- OUTPUT:
  - salary = 65000.0
  - vacation days = 21
  - vacation form = pink

Polymorphism + arrays

- You can declare arrays of superclass types, and store objects of any subtype as elements.

```java
public class EmployeeMain2 {
    public static void main(String[] args) {
        Employee[] employees = {new Lawyer(3), new Secretary(2),
                                new Marketer(4), new LegalSecretary(1)};
        for (int i = 0; i < employees.length; i++) {
            System.out.println("salary = " + employees[i].getSalary());
            System.out.println("vacation days = " + employees[i].getVacationDays());
            System.out.println();
        }
    }
}
```

- OUTPUT:
  - salary = 65000.0
  - vacation days = 21
  - vacation form = pink
  - salary = 50000.0
  - vacation days = 10
  - vacation form = yellow

Polymorphism problems

- The textbook has several useful exercises to test your knowledge of polymorphism.
  - Each exercise declares a group of approximately 4 or 5 short classes with inheritance is-a relationships between them.
  - Then a client program is shown that calls methods on objects of each class.
  - Your task is to interpret the code and determine the output of the client program.

(Example on next slide...)
A polymorphism problem

- Assume that the following four classes have been declared:

  ```java
  public class Foo {
    public void method1() {
      System.out.println("foo 1");
    }
    public void method2() {
      System.out.println("foo 2");
    }
    public String toString() {
      return "foo";
    }
  }
  public class Bar extends Foo {
    public void method2() {
      System.out.println("bar 2");
    }
    public String toString() {
      return "baz";
    }
  }
  public class Mumble extends Baz {
    public void method2() {
      System.out.println("mumble 2");
    }
    public String toString() {
      return "baz";
    }
  }
  public class Baz extends Foo {
    public void method1() {
      System.out.println("baz 1");
    }
    public void method2() {
      System.out.println("baz 2");
    }
    public String toString() {
      return "baz";
    }
  }
  ```

- 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();
  }
  ```

Finding output with diagrams

- One way to determine the output is to diagram each class and its methods, including their output:
  - Add the classes from top (superclass) to bottom (subclass).
  - Include any inherited methods and their output.

![Diagram of class hierarchy]

Finding output with tables

- Another possible technique for solving these problems is to make a table of the classes and methods, writing the output in each square.

<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>foo</td>
<td>baz</td>
<td>baz</td>
</tr>
</tbody>
</table>

Polymorphism answer

- The code produces the following output:
  ```java
  baz
  baz 1
  foo 2
  foo
  foo 1
  bar 2
  baz
  baz 1
  mumble 2
  foo
  foo 1
  foo 2
  ```
Another problem

Assume following classes have been declared. The order of classes is changed, as well as the client.

```java
public class Lamb extends Ham {
    public void b() {
        System.out.println("Lamb b");
    }
}
```

```java
public class Spam extends Yam {
    public void a() {
        System.out.println("Spam a");
    }
}
```

What would be the output of the following client code?

```java
Ham[] food = {new Spam(), new Yam(), new Ham(), new Lamb();}
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    food[i].b();
    System.out.println();
}
```

The class diagram

The following diagram depicts each class's behavior:

The table

The following table also depicts each class's behavior:

<table>
<thead>
<tr>
<th>method</th>
<th>Ham</th>
<th>Lamb</th>
<th>Yam</th>
<th>Spam</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>Ham a</td>
<td>Ham a</td>
<td>Yam a</td>
<td>Spam a</td>
</tr>
<tr>
<td>b</td>
<td>Ham b</td>
<td>Lamb b</td>
<td>Lamb b</td>
<td>Lamb b</td>
</tr>
<tr>
<td>toString</td>
<td>Ham</td>
<td>Ham</td>
<td>Yam</td>
<td>Yam</td>
</tr>
</tbody>
</table>

The answer

```java
Ham[] food = {new Spam(), new Yam(), new Ham(), new Lamb();}
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    food[i].b();
    System.out.println();
}
```

The code produces the following output:

Yam
Spam a
Lamb b
Yam
Yam a
Lamb b
Ham
Ham a
Ham b
Ham
Ham a
Lamb b
Relatedness of types

Consider the task of writing classes to represent 2D shapes such as Circle, Rectangle, and Triangle.

- There are certain attributes or 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 these attributes, but each 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

- Let's write shape classes with methods named \(\text{perimeter}\) and \(\text{area}\).

- We'd like to be able to write client code that treats different shape objects in the same way, insofar as they share common behavior, 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 \text{DrawingPanel} display many shapes on screen.
Interfaces

- **interface**: A list of methods that classes can promise to 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 non-programming idea of roles or certifications
    - "I'm certified as a CPA accountant. The certification assures you that I know how to do taxes, perform audits, and do management consulting."
    - "I'm certified as a Shape. That means you can be sure that I know how to compute my area and perimeter."

Interface syntax

- **Interface declaration, general syntax**:
  ```java
  public interface <name> {
    public <type> <name>(<type> <name>, ..., <type> <name>);
    public <type> <name>(<type> <name>, ..., <type> <name>);
    ...
    public <type> <name>(<type> <name>, ..., <type> <name>);
  }
  ```
  **Example:**
  ```java
  public interface Vehicle {
    public double getSpeed();
    public void setDirection(int direction);
  }
  ```

Abstract method: A method header without an implementation.
- The actual bodies of the methods are not specified, because we want to allow each class to implement the behavior in its own way.
- **Exercise**: Write an interface for shapes.

Shape interface

- **An interface for shapes**:
  ```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

- A class can declare that it **implements** an interface.
  - This means the class contains an implementation for each of the abstract methods in that interface.
    (Otherwise, the class will fail to compile.)

- **Implementing an interface, general syntax**:
  ```java
  public class <name> implements <interface name> {
    ...
  }
  ```
  **Example:**
  ```java
  public class Bicycle implements Vehicle {
    ...
  }
  ```
  (What must be true about the **Bicycle** class for it to compile?)
Interface requirements

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

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

Diagrams of interfaces

- We draw arrows upward from the classes to the interface(s) they implement.
  - 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.

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 and polymorphism

- Using interfaces doesn't benefit the class author so much as the client code author.
  - The is-a relationship provided by the interface means that the client can take advantage of polymorphism.
  - Example:
    ```java
    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 as the parameter to the above method.
      ```java
      Circle circ = new Circle(12.0);
      Triangle tri = new Triangle(5, 12, 13);
      printInfo(circ);
      printInfo(tri);
      ```

Arrays of interface type

- We can create an array of an interface type, and store any object implementing that interface as an element.
  ```java
  Circle circ = new Circle(12.0);
  Rectangle rect = new Rectangle(4, 7);
  Triangle tri = new Triangle(5, 12, 13);
  Shape[] shapes = {circ, tri, rect};
  for (int i = 0; i < shapes.length; i++) {
      printInfo(shapes[i]);
  }
  ```
- Each element of the array executes the appropriate behavior for its object when it is passed to the printInfo method, or when area or perimeter is called on it.