CS 241 — Introduction to Problem Solving and Programming

Object-Oriented Programming

Abstract classes and methods

April 1, 2005

What's left before the next test

(Today)

I. Subtyping and interface inheritance

II. Abstract classes and methods

III. Subclassing relations

(Monday)

IV. Method overriding

V. Other subclassing details

(Wednesday)

VI. The class hierarchy

VII. Dynamic binding

(Friday)

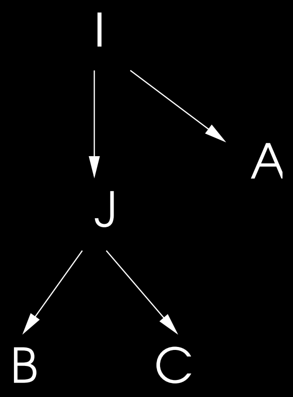
Review and lab day

(Monday)

TEST

Review(ish)

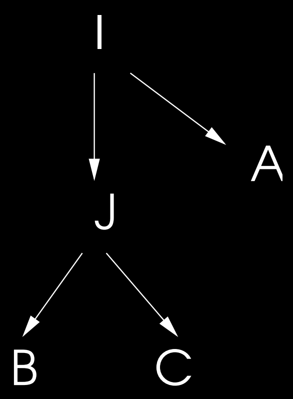
One interface may extend another. The extending interface implicitly contains all the method signatures from the extended one; they do not need to be named.



$$\begin{tikzcd}
 & I \arrow[d] \arrow[r] & A \\
 & J \arrow[dl] \arrow[dr] & \\
B & & C
\end{tikzcd}$$

```
interface I {  ... }

class A implements I { ... }

interface J extends I { ... }

class B implements J { ... }
class C implements J { ... }
```

Review(ish)


$$\begin{tikzcd} I \arrow[d] \arrow[dr] & \\ J \arrow[d] \arrow[dr] & A \\ B & C \end{tikzcd}$$

```
interface I {  ... }

class A implements I { ... }

interface J extends I { ... }

class B implements J { ... }
class C implements J { ... }
```

Here, `B` is a subtype of `J`, which is a subtype of `I`.

An instance of `B` *is a* `J` and it *is an* `I`.

Review(ish)

$$\begin{tikzcd} & I \arrow[d] \arrow[dr] & \\ & J \arrow[dl] \arrow[dr] & A \\ B & & C \end{tikzcd}$$

```
interface I {  ... }

class A implements I { ... }

interface J extends I { ... }

class B implements J { ... }
class C implements J { ... }
```

Class `A` must provide implementations for all of `I`'s methods.

Class `B` must provide implementations for all of `I`'s methods *and* all of `J`'s methods.

Function example

```
interface Function {
    public double evaluate(double x);
}

class Floor implements Function {
    ...
}

interface Differentiable extends Function {
    public Function derivative();
}

class Polynomial implements Differentiable {
     public double evaluate(double x) { ... }
     public Function derivative() { ... }
}
```

Review(ish)

$$\begin{tikzcd}
I \arrow[d] \arrow[dr] & \\
J \arrow[d] \arrow[dr] & A \\
B & C
\end{tikzcd}$$

```
interface I {  ... }

class A implements I { ... }

interface J extends I { ... }

class B implements J { ... }
class C implements J { ... }
```

In terms of sets of objects, B $\subseteq$ J $\subseteq$ I.

Review(ish)

$$\begin{tikzcd}
 & I \arrow[d] \arrow[dr] & \\
 & J \arrow[dl] \arrow[dr] & A \\
B & & C
\end{tikzcd}$$

```
interface I {  ... }

class A implements I { ... }

interface J extends I { ... }

class B implements J { ... }
class C implements J { ... }
```

In terms of sets of methods, $\mathtt{I} \subseteq \mathtt{J} \subseteq \mathtt{B}$

Inheritance

An interface `J` that extends another interface `I` inherits all the method signatures of `I`.

Inheritance is the passing on of properties from a base type or structure to a derived type or structure.

Inheritance is very important for object-oriented programming, specifically for abstraction and reusability of code.

Abstraction

General problem:

You are writing two classes `A` and `B` which have a common subset of public methods (ie, interface) and which you want to use interchangeably. Accordingly, you define an interface `I` which `A` and `B` implement.

However, `A` and `B` share more attributes. Much of their data (ie, instance variables) is the same, and some of the functionality (methods) are also the same.

Abstraction

Specific problem (example):

You are writing a payroll program. The company has both salaried and hourly employees, which you plan to model separate classes. They share an interface (print directory entry, calculate pay. . .).

Payroll

```
public interface Employee {

     public void printDirectory();

     public double computePay();
}
```

Payroll

```
public class Hourly implements Employee {
     private String name;
     private int officeNumber;
     private String building;
     private double rate;
     private double hours;
     public printDirectory() {
          System.out.println(name + " " + officeNumber + "" building);
     }
     public double computePay() {
          double pay = hours * rate;
          hours = 0;
          return pay;
     }
}
```

Payroll

```
public class Salaried implements Employee {
     private String name;
     private int officeNumber;
     private String building;
     private double salary;

     public printDirectory() {
          System.out.println(name + " " + officeNumber + "" building);
     }
     public double computePay() {
          return salary / 26;
     }
}
```

Reuse

Redundancy is bad. Reuse is good.

Reuse

Redundancy is bad. Reuse is good.

- Convenience
- Mistake tracing
- Maintenance

Abstract classes and methods

The solution is abstract classes.

An abstract class is halfway between an interface and a non-abstract class. It can declare instance variables and define methods. But it can also declare method signatures without bodies (or inherit them from an interface).

Methods that are declared but not defined are abstract methods. (Alternately, an abstract class is a class that has abstract methods.)

An abstract class cannot be instantiated.

Abstract classes and methods

```
public interface I {
     public int m(double x);
     public void n(String str);
}
public abstract class A implements I {
     private int z;
     public int m(double x) { return (int) z * x; }
     public abstract int mm();
}
public class C extends A {
     private String ss;
     public void n(String str) { ss = str; }
     public int mm() { return m() + ss.length(); }
}
```

Abstract classes and methods

```
public interface I {
     public int m(double x);
     public void n(String str);
}
public abstract class A implements I {
     private int z;
     public int m(double x) { return (int) z * x; }
     public abstract int mm();
}
public class C extends A {
     private String ss;
     public void n(String str) { ss = str; }
     public int mm() { return m() + ss.length(); }
}
```

`A` implements `I`. It can implement some of `I`'s methods; any it does not implement are implicitly abstract.

Abstract classes and methods

```
public interface I {
     public int m(double x);
     public void n(String str);
}
public abstract class A implements I {
     private int z;
     public int m(double x) { return (int) z * x; }
     public abstract int mm();
}
public class C extends A {
     private String ss;
     public void n(String str) { ss = str; }
     public int mm() { return m() + ss.length(); }
}
```

`A` declares an abstract class `mm`, which all child classes must implement.

Abstract classes and methods

```
public interface I {
     public int m(double x);
     public void n(String str);
}
public abstract class A implements I {
     private int z;
     public int m(double x) { return (int) z * x; }
     public abstract int mm();
}
public class C extends A {
     private String ss;
     public void n(String str) { ss = str; }
     public int mm() { return m() + ss.length(); }
}
```

`C` extends `A`, which means it inherits the defined methods and must also implement the abstract methods.

Abstract classes and methods

```
public interface I {
     public int m(double x);
     public void n(String str);
}
public abstract class A implements I {
     private int z;
     public int m(double x) { return (int) z * x; }
     public abstract int mm();
}
public class C extends A {
     private String ss;
     public void n(String str) { ss = str; }
     public int mm() { return m() + ss.length(); }
}
```

A is a subtype of I. C is a subtype of A, and in turn a subtype of I.

Payroll

```
public interface Employee {
     public void printDirectory();
     public double computePay();
}

public abstract class AbstractEmployee
          implements Employee {
     private String name;
     private int officeNumber;
     private String building;
     public printDirectory() {
          System.out.println(name + " " +
               officeNumber + "" building);
     }
}
```

```
public class Hourly extends AbstractEmployee {
     private double rate;
     private double hours;
     public double computePay() {
          double pay = hours * rate;
          hours = 0;
          return pay;
     }
}
public class Salaried extends AbstractEmployee {
     private double salary;
     public double computePay() {
          return salary / 26;
     }
}
```

Payroll

```
public abstract class Employee {
     private String name;
     private int officeNumber;
     private String building;
     public printDirectory() {
          System.out.println(name + " "
             + officeNumber + "" building);
     }
     public abstract double computePay();
}
```

```
public class Hourly extends Employee {
     private double rate;
     private double hours;
      public double computePay() {
          double pay = hours * rate;
          hours = 0;
          return pay;
     }
}
public class Salaried extends Employee {
     private double salary;
     public double computePay() {
          return salary / 26;
     }
}
```

Inheritance

Keep extends and implements straight.

If (non-abstract) a class implements an interface, it must implement all the methods mentioned in the interface.

If an interface extends another interface, it inherits the method signatures.

If an abstract class implements an interface, it or its subclasses must implement all the methods mentioned in the interface; it inherits the methods in the interface which it does not implement as abstract.

If a class extends another class, it inherits all the members of that class and must implement any abstract methods.

Although we have seen that a class may implement more than one interface, Java does not allow a class to extend more than one class (called multiple inheritance).

Inheritance

What about instance variables?

They are inherited, too.

```
public abstract class A {
     int x;
     abstract int f(int y);
}

public class C extends A {
     int f(int y) {
          return y * x;
     }
}
```

Inheritance

What about access level?

Private instance variables (and methods) are accessibly *only* to the class in which they are declared, not even subclasses.

```
public abstract class A {
     private int x;
     public abstract int f(int y);
}

public class C extends A {
     public int f(int y) {
          return y * x;        // Error
     }
}
```

Inheritance

One solution is to use the `protected` access modifier. This makes members accessible to the class and all subclasses.

```
public abstract class A {
     protected int x;
     public abstract int f(int y);
}

public class C extends A {
     public int f(int y) {
          return y * x;       // OK
     }
}
```

Payroll

```
public abstract class Employee {
     private String name;
     private int officeNumber;
     private String building;
     protected double payToDate;
     public printDirectory() {
          System.out.println(name + " "
             + officeNumber + "" building);
     }
     public abstract double computePay();
}
```

```
public class Hourly extends Employee {
     private double rate;
     private double hours;
      public double computePay() {
          double pay = hours * rate;
          hours = 0;
          payToDate += pay;
          return pay;
     }
}
public class Salaried extends Employee {
     private double salary;
     public double computePay() {
          double pay = salary / 26;
          payToDate += pay;
          return pay;
     }
}
```

Access modifiers

Summary of access modifiers

	Modifier	Accessibility of member
highest	`public`	All other classes
	`protected`	Subclasses and other classes in the same package
	(default)	Other classes in the same package
lowest	`private`	No other classes

Access modifiers

When to use. . .

`public` All methods you intend other classes (including ones with main methods) to call.

`protected` Instance variables and methods you otherwise would want `private` but *really need* available to subclasses (avoid: better solution usually to require subclasses to use methods, possibly getter and setter).

default Never? Possibly on short experimental classes. . . never in finished, ready-to-turn-in product.

`private` All instance variables; any method that outside classes have no need for.

Payroll

```
public abstract class Employee {
     private String name;
     private int officeNumber;
     private String building;
     private double payToDate;
     protected double reportPay(double pay) {
          payToDate += pay;
          return pay;
     }
     public printDirectory() {
          System.out.println(name + " "
             + officeNumber + "" building);
     }
     public abstract double computePay();
}
```

```
public class Hourly extends Employee {
     private double rate;
     private double hours;
      public double computePay() {
          double pay = hours * rate;
          hours = 0;
          return reportPay(pay);
     }
}
public class Salaried extends Employee {
     private double salary;
     public double computePay() {
          return reportPay(salary / 26);
     }
}
```

Extension

A class does not have to be abstract in order for a class to extend it.

```
public interface Shape {
     public double area();
}
public class Triangle {
     protected double base;
     protected double height;
     public double area() { return base * height; }
}
public class RightTriangle extends Triangle {
     public double hypotenuse() {
          return Math.sqrt(base*base + height*height);
     }
}
```

Inheritance

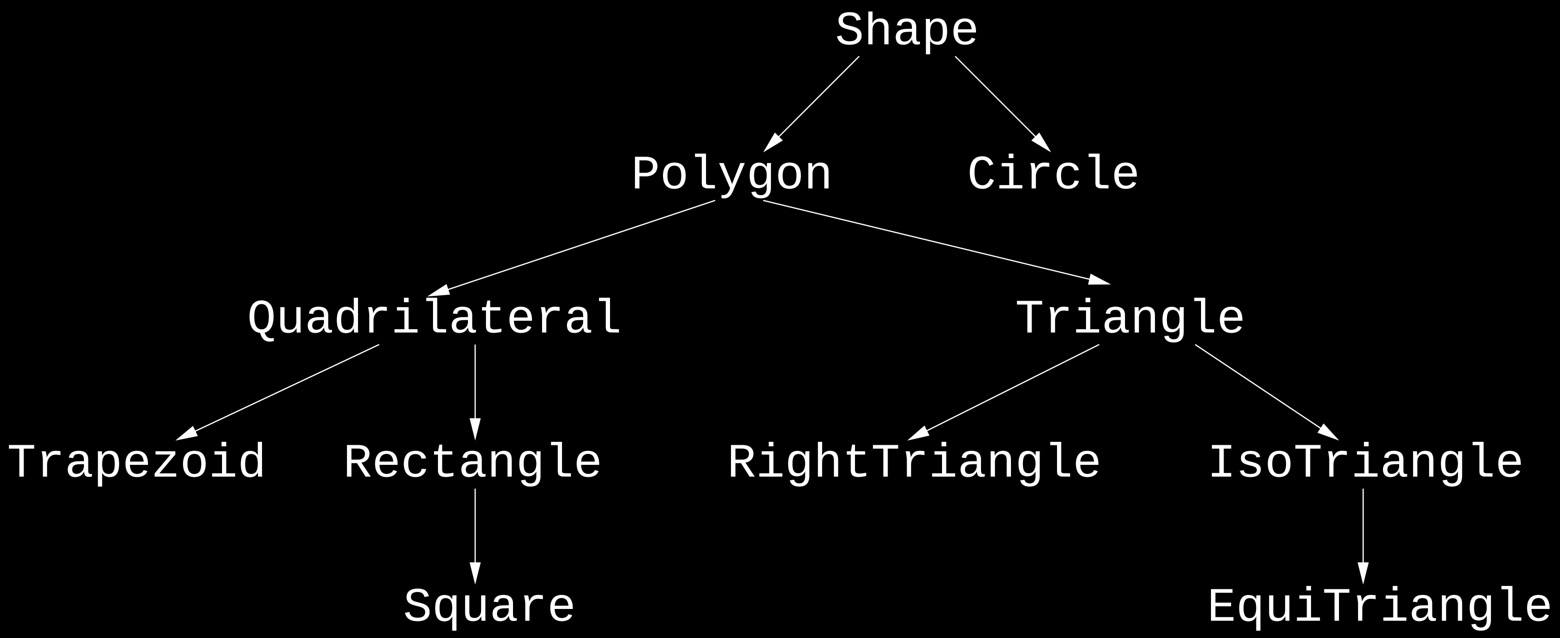
Find the errors.

```
public interface I {
     public int m(double x);
}

public abstract class A extends I {
     private int z;
     public abstract void n(String str);
     public int m(double x) {
          return Math.ceil(z + x);
     }
}

public class B extends A {
     public abstract double p(double d);
     public void n(String str) {
          System.out.println(str);
     }
}
```

```
public class C implements B {
     private int y;
     public double p(double d) {
          return (z + y) / d;
     }
}

public class T {
     public static void main(String[] args) {
          I i, j;
          i = new A();
          System.out.println(i.m(1.2));
          j = new C();
          System.out.println(j.p(1.2));
     }
}
```

Relations and terms

Designing a class hierarchy is a fundamental part of developing a piece of software.



Relations and terms

Suppose we have classes (any possibly abstract) `A`, `B`, and `C`.

`A` is the parent class of `B`. `B` is the parent class of `C`. `B` is the child class of `A`. `C` is the child class of `B`.

`B` and `C` are descendent classes of `A`. `A` and `B` are ancestor classes of `C`.

`A` and `B` are base classes. `B` and `C` are derived classes.

Exercise

Design a class hierarchy for a library program.

- All library items have titles and publishers.
- Periodicals do not have call numbers.
- Periodicals and reference books may not be checked out.
- All books have page numbers.
- Recordings and books have publishers.
- Recordings and non-reference books have authors.

Summary

- Extend
- Implement
- Inheritance
- Base/derived type
- Why redundancy is bad and reuse is good
- Abstract classes and methods
- Access modifiers (public, private, protected)
- Class hierarchy
- Parent/child, ancestor/descendent classes