CS 241 — Introduction to Problem Solving and Programming

Object-Oriented Programming

More on inheritance; method overriding

April 4, 2005

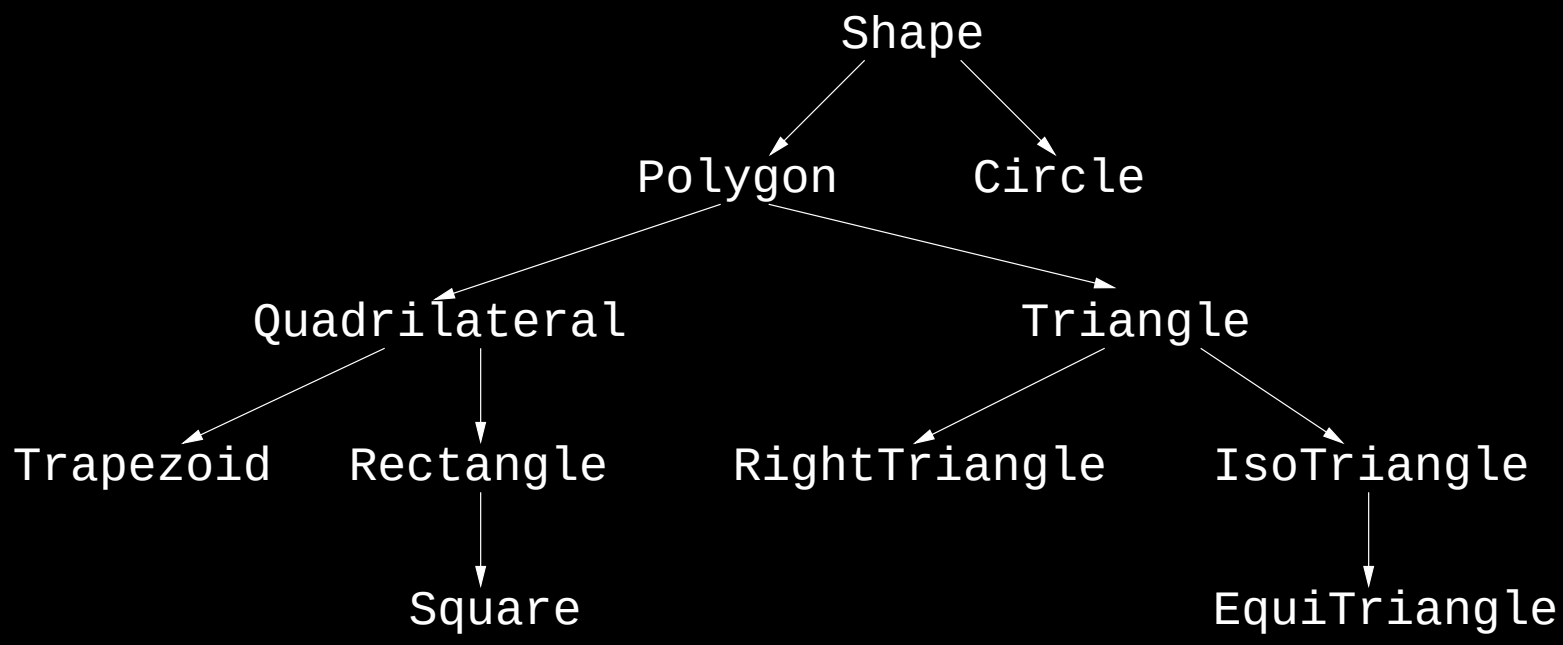
Extension

A class my extend another class, in which case it inherits all of its members and produces a subtype of the original class's type.

```
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);
     }
}
```

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

How we got here

We have looked at concepts in this order:

- Classes
- Interfaces
- Abstract classes and inheritance
- Method overriding

Our textbook introduces concepts in this order:

- Classes
- Inheritance
- Method overriding
- Abstract classes
- Interfaces

Problem

What if we wanted to inherit some methods, but not all—perhaps some methods should be defined differently in a child class.

Example:
Recall payroll example. Suppose some hourly employees should be paid time and a half for overtime.

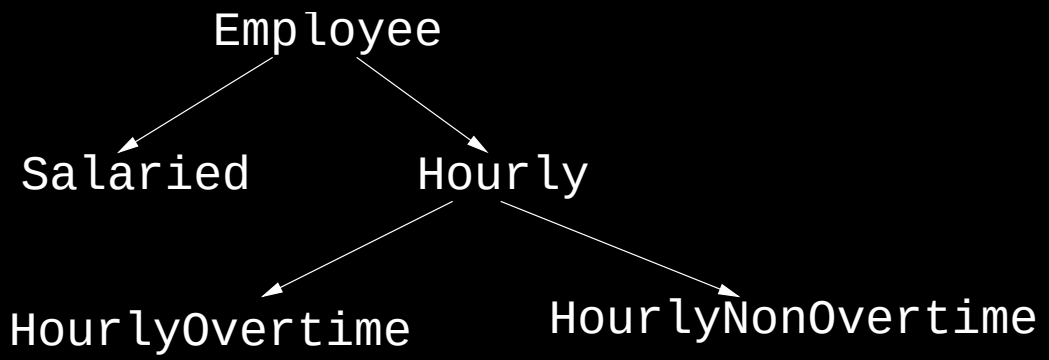
Payroll

```
public class Hourly extends Employee {
     private double rate;
     private double hours;

     public void setRate(double rate) {
          this.rate = rate;
     }

     public double computePay() {
          double pay = hours * rate;
          hours = 0;
          return reportPay(pay);
     }
}
```

The desired new class, `HourlyOvertime` should be an `Hourly` and inherit everything, but pay should be computed differently.

Payroll

One solution is to add another level of abstraction, keep `computePay()` abstract in `Hourly`, and add two classes.



Payroll

Another solution: Keep the old hierarchy; make `HourlyOvertime` a subclass of `Hourly`, and simply redefine `computePay()`

```
public class HourlyOvertime extends Hourly {
     public double computePay() {
          double pay;
          if (hours <= 40)
               pay = hours * rate;
          else
               pay = (40 * rate) + ((hours - 40) * 1.5 * rate);
          hours = 0;
          return reportPay(pay);
     }
}
```

Java allows a subclass to implement a parent class's method even if it is not abstract.

Overriding

When a child class redefines a parent class's method, it is called method overriding.

This allows a child class to selectively inherit methods, or say to the parent class, "No thanks, I'll implement this one myself."

When that method is called on an instance of the child class, the method defined in the child class is executed instead of that in the parent class.

Overriding

```
public abstract class A {
     public int m1(int y);
}

public class B extends A {
     private int x;
     public int m1(int y) {  return x * y; }
     public int m2(int z) {  return x - z; }
}

public class C extends B {
     private int xx;
     public int m2(int z) { return xx - z; }
}
```

```
A a = new B();
a.m1();
a.m2();
a = new C();
a.m1();
a.m2();
```

Overriding

```
public abstract class A {
     public int m1(int y);
}

public class B extends A {
     private int x;
     public int m1(int y) {  return x * y; }
     public int m2(int z) {  return x - z; }
}

public class C extends B {
     private int xx;
     public int m2(int z) { return xx - z; }
}
```

```
A a = new B();
a.m1();
a.m2();
a = new C();
a.m1();
a.m2();
```

Overriding

```
public abstract class A {
     public int m1(int y);
}

public class B extends A {
     private int x;
     public int m1(int y) {  return x * y; }
     public int m2(int z) {  return x - z; }
}

public class C extends B {
     private int xx;
     public int m2(int z) { return xx - z; }
}
```

```
A a = new B();
a.m1();
a.m2();
a = new C();
a.m1();
a.m2();
```

Overriding

```
public abstract class A {
     public int m1(int y);
}

public class B extends A {
     private int x;
     public int m1(int y) {  return x * y; }
     public int m2(int z) {  return x - z; }
}

public class C extends B {
     private int xx;
     public int m2(int z) { return xx - z; }
}
```

```
A a = new B();
a.m1();
a.m2();
a = new C();
a.m1();
a.m2();
```

Overriding

```
public abstract class A {
     public int m1(int y);
}

public class B extends A {
     private int x;
     public int m1(int y) {  return x * y; }
     public int m2(int z) {  return x - z; }
}

public class C extends B {
     private int xx;
     public int m2(int z) { return xx - z; }
}
```

```
A a = new B();
a.m1();
a.m2();
a = new C();
a.m1();
a.m2();
```

Overriding

```
public abstract class A {
     public int m1(int y);
}

public class B extends A {
     private int x;
     public int m1(int y) {  return x * y; }
     public int m2(int z) {  return x - z; }
}

public class C extends B {
     private int xx;
     public int m2(int z) { return xx - z; }
}
```

```
A a = new B();
a.m1();
a.m2();
a = new C();
a.m1();
a.m2();
```

Overriding

```
public abstract class A {
     public int m1(int y);
}

public class B extends A {
     private int x;
     public int m1(int y) {  return x * y; }
     public int m2(int z) {  return x - z; }
}

public class C extends B {
     private int xx;
     public int m2(int z) { return xx - z; }
}
```

```
A a = new B();
a.m1();
a.m2();
a = new C();
a.m1();
a.m2();
```

Overriding and overloading

It is critical not to confuse overriding with overloading.

Despite their similar names, they are quite different things.

Overriding and overloading

It is critical not to confuse overriding with overloading.

Despite their similar names, they are quite different things.

Overloading is when a class has more than one method with the same name but different signatures.

Overriding is when a subclass has a method of the same name and signature as a method in the parent class.

For a given invocation of an overloaded method, when is it disambiguated, compile time or runtime? What about for an overridden method?

Overriding and overloading

If a method is overloaded, an invocation is disambiguated at compile time based on the static types of its parameters.

If a method is overridden, an invocation is disambiguated at run time based on the dynamic type of its receiver.

Details

`super`, `this`, and `final`

Super

```
public class Hourly extends Employee {
     private double rate;
     private double hours;

     public void setRate(double rate) {
          this.rate = rate;
     }

     public double computePay() {
          double pay = hours * rate;
          hours = 0;
          return reportPay(pay);
     }
}
```

```
public class HourlyOvertime extends Hourly {
    public double computePay() {
           double pay;
          if (hours <= 40)
                pay = hours * rate;
          else
               pay = (40 * rate) +
                     ((hours - 40) * 1.5 * rate);
           hours = 0;
          return reportPay(pay);
     }
}
```

This seems redundant. The child class's computePay() uses the parent class's code, merely adding to it.

Super

Java allows you to call a parent class's method by prefixing the word `super`.

```
super.method(params);
```

`super` is essentially like `this`, except that it uses the current instance as a instance of its parent class.

Super

```
public class Hourly extends Employee {
     private double rate;
     private double hours;

     public void setRate(double rate) {
          this.rate = rate;
     }

     public double computePay() {
          double pay = hours * rate;
          hours = 0;
          return reportPay(pay);
     }
}
```

```
public class HourlyOvertime extends Hourly {
     public double computePay() {
          double overtimePay = 0;
          if (hours > 40) {
               overtimePay = (hours - 40) * 1.5 * rate;
               reportPay(overtimePay);
               hours = 40;
          }
          return overtimePay + super.computePay();
     }
}
```

Super

Java does not allow the chaining of super.

```
public class Adder {
     protected int z;
     public int m(int x) { return x + z; }
}
public class Multiplier extends Adder {
     public int m(int x) { return x * z; }
}
public class Both extends Multiplier {
     public int m(int x) {
          return super.m()
               - super.super.m();     // error!
     }
}
```

Constructors

We've said that it is best to leave instance variables in parent classes `private` (as opposed to `protected`).

How then can we initialize those instance variables when instantiating a child class?

```
public class A {
     private int a;
     public A(int a) {
          this.a = a;
     }
}
public class B extends A {
     private int b;
     public B(int a, int b) {
          this.b = b;
          this.a = a;     // error-- a is private.
                          // how else can I initialize a?
     }
}
```

Constructors

You may call a constructor of a super class from the constructor of a child class.

`super` followed by a parameter list in parentheses invokes a parent class's constructor.

```
public class A {
     private int a;
     public A(int a) {
          this.a = a;
     }
}
public class B extends A {
     private int b;
     public B(int a, int b) {
          this.b = b;
          super(a);
     }
}
```

Constructors

While we're on the topic of constructors. . .

```
public class SetString {
     private String elements;
     public Set() {
          elements = "";
     }
     public Set(char initial) {
          elements = "" + initial;
     }
}
```

Constructors

While we're on the topic of constructors. . .

```
public class SetString {
     private String elements;
     public Set() {
          elements = "";
     }
     public Set(char initial) {
          this();
          elements += initial;
     }
}
```

Final

You may also declare classes, instance variables, and methods `final`.

A `final` class cannot be extended (subclassed).

A `final` method may not be overridden.

A `final` instance variable may not be assignment more than once (initialized value stays forever).

Summary

Understand these concepts.

- Method overriding
- Method overloading
- `super`
- `super` and
- this as constructor calls
- `final`