CS 241 — Introduction to Problem Solving and Programming

Object-Oriented Programming

Subtype Polymorphism, part II

Mar 30, 2005

Review

- Type
- Subtype
- Interface
- Class
- Static type
- Dynamic type

Review

```
public interface I {
    public int m(double x);
    public char n(String st, int y);
}

public class C implements I{
    double z;
    public C(double z) {  this.z = z; }
    public int m(double x) { return (int) z - x;}
    public char n(String st, int y) {
        if (st.length() > (z / y)) return 'a';
        else return 'b';
    }
}

public class D implements I { ...
```

```
{
    I i = new C(15.7);
    int q = i.m(12.3);
    i = new D();
    S.out.pl(i.n("hi", 2) + q);
}
```

Point

Realistic example: Points in Cartesian or polar coordinates.

```
public interface Point {
     public double xCoord();
     public double yCoord();
     public double distance();
     public double radians();
}
```

Point

```
public class Cartesian implements Point {
     private double x;
     private double y;
     public double xCoord() { return x; }
     public double yCoord() { return y; }
     public double distance() { return Math.sqrt(x*x + y*y); }
     public double radians() {
          int n = 0;
          if (x >= 0)
               if (y == 0) n = .5;
               else if (y > 0) n = 1;
          return Math.atan(y/x) + n;
     }
}
```

Point

```
public class Polar implements Point {
     public double r;
     public double theta;
     public double xCoord() { return r * Math.cos(theta); }
     public double yCoord() { return r * Math.sin(theta); }
     public double distance() { return r; }
     public double radians() { return theta; }
}
```

Messages

Think of a method invocation as sending a message to an object.

An alternate name for *invocation* is *message send*.

The return value is the object's response to the message. For void methods, the object responds by doing something but does not return a value.

How then can we think of classes and interfaces?

Messages

Think of a method invocation as sending a message to an object.

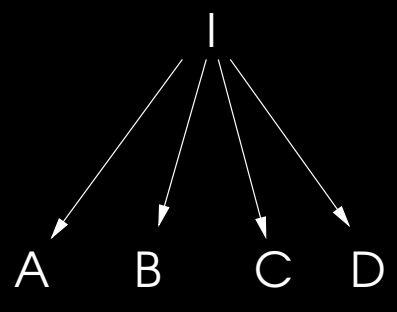
"Do *this* (method name) with *these* (arguments)."

An interface lists what messages (as a minimum) an object understands; however, different objects might understand the same messages differently.

A class defines what an object does with them.

Subtyping

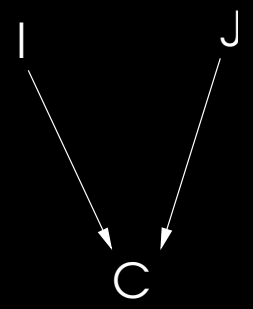
So far we have seen only two levels of subtyping: interface and classes.




```
interface I {  ... }

class A implements I { ... }
class B implements I { ... }
class C implements I { ... }
class D implements I { ... }
```

Subtyping

A class may implement more than one interface.




```
interface I {  ... }

interface J { ... }

class C implements I,J { ... }
```

Multiple implementation

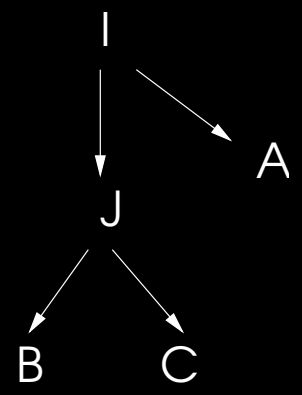
```
interface Student {
     public int year();
     public double GPA();
}
```

```
interface Employee {
     public double pay(double hours);
     public int withholdings();
}
```

```
public class  StudentEmployee implements Employee, Student{
     public double pay(double hours) { ... }
     public int withholdings() { ... }
     public int year() { ... }
     public double GPA() { ... }
}
```

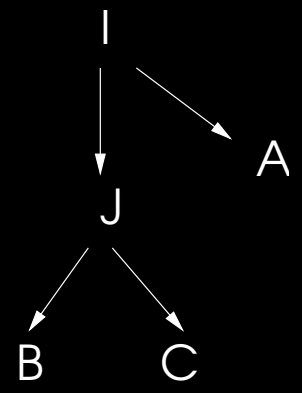
Extending interfaces

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.




```
interface I {  ... }

class A implements I { ... }

interface J extends I { ... }

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

Extending interfaces

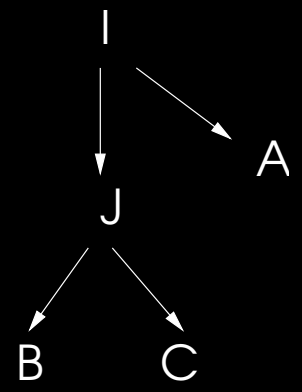



```
interface I {  ... }

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.

Extending interfaces




```
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() { ... }
}
```

Instanceof

Static vs dynamic types

You can *test* an expression to determine a dynamic type.

```
x instanceof C
```

`C` is a reference type. The expression has type `boolean`

This *gains* information—how might information have been lost?

Cast

Static vs dynamic types

You can *cast* an expression if you know its dynamic type.

```
((C) x).m()
```

Cast

```
if (f instanceof Differentiable)
   return ((Differentiable) f).derivative();
```

Summary

- Type
- Subtype
- Polymorphism
- Methods as message send
- Dynamic binding
- `instanceof`
- Casts