ACCESS CONTROL, CODE REUSE, INHERITANCE, POLYMORPHISM

17TH LECTURE

엄현상(Eom, Hyeonsang)
School of Computer Science and Engineering
Seoul National University

Outline

- **Access Control**
  - Access Control
  - Class Access
- **Code Reuse**
  - Reusing Class
- **Inheritance**
  - Composition vs Inheritance
  - Composition syntax
  - Inheritance syntax
  - final
- **Polymorphism**
  - Interface & Implementation
  - Dynamic Binding
  - Abstract classes
  - Constructors
  - Pure Inheritance vs Extension

Java Access Control

- Introduction
- Function Templates
- Overloading Function Templates
- Class Templates
- Nontype Parameters and Default Types for Class Templates

Java Access Control

public	Interface Access
private	Only Accessible Within the class
protected	“Sort of private” deals with inheritance

“Friendly”

- Default access : no keyword
- **Public**
  - Other members of the same package
- **Private**
  - Anyone outside the package
  - Easy interaction for related classes (that you place in the same package)
  - Also referred to as “package access”

Protected

- **protected**
  - Inheritors (and the package) can access protected members

  BUT
  - They are then vulnerable to changes in the base-class implementation
  - Users of classes not in the class hierarchy are prevented from accessing protected members

public: Interface Access

```
package c05.dessert;

public class Cookie {
  public Cookie() {
    System.out.println("Cookie
constructor");
  }
  void bite()
{ System.out.println("bite"); }
} ///:~
```

```
//: c05:Dinner.java
// Uses the library.
import c05.dessert.*;

public class Dinner {
  public Dinner() {
    System.out.println("Dinner
constructor");
  }
  public static void main(String[]
args) {
    Cookie x = new Cookie();
    //! x.bite(); // Can't access
  }
} ///:~
```

private: Can’t Touch That!

```
//: c05:IceCream.java
// Demonstrates "private" keyword.

class Sundae {
  private Sundae() {}
  static Sundae makeASundae() {
    return new Sundae();
  }
}

public class IceCream {
  public static void main(String[] args) {
    //! Sundae x = new Sundae();
    Sundae x = Sundae.makeASundae();
  }
} ///:~
```

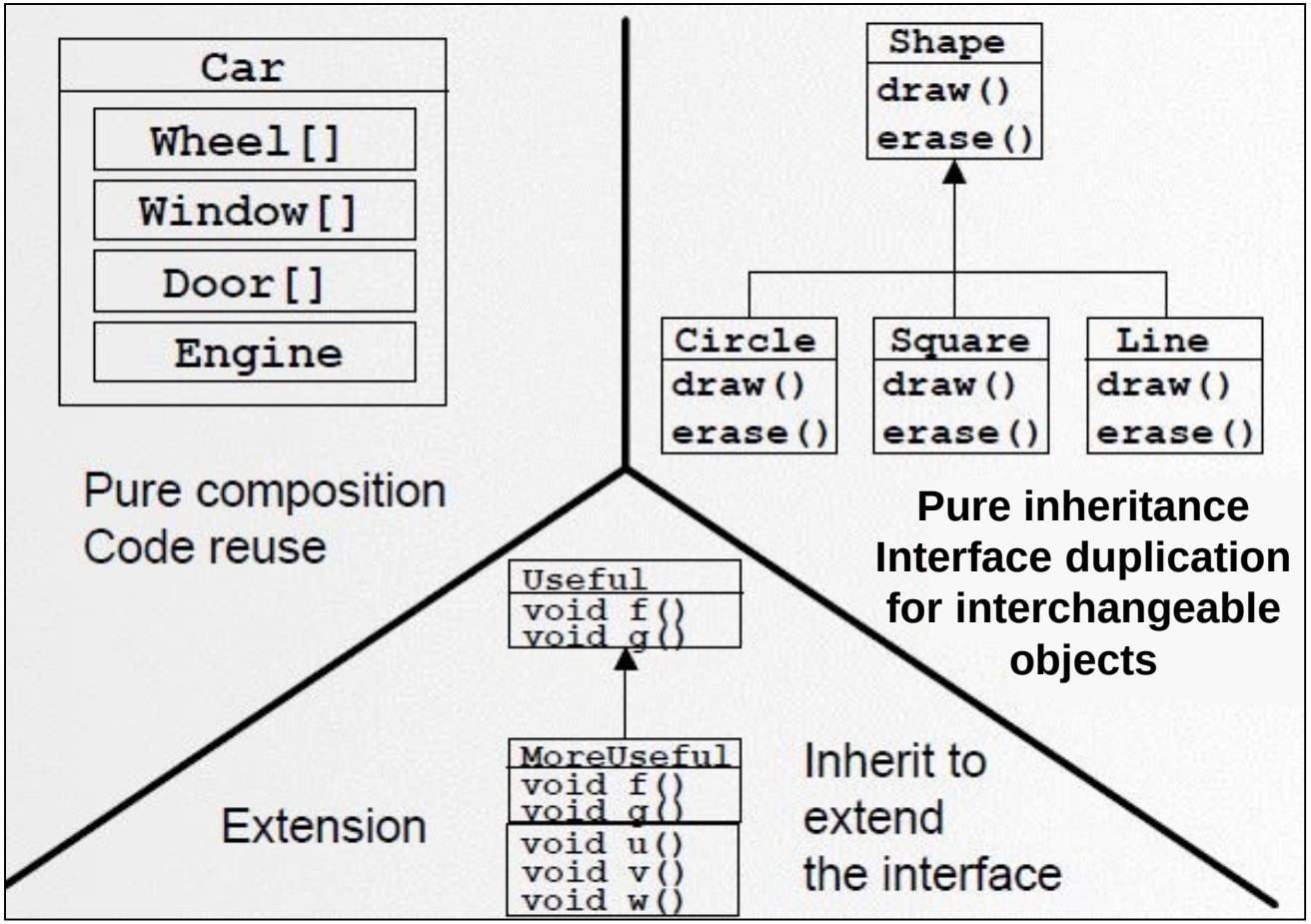
Class Access

- Classes as a whole can be **public** or "friendly"
- Only one **public** class per file, usable outside the package
- All other classes "friendly," only usable within the package

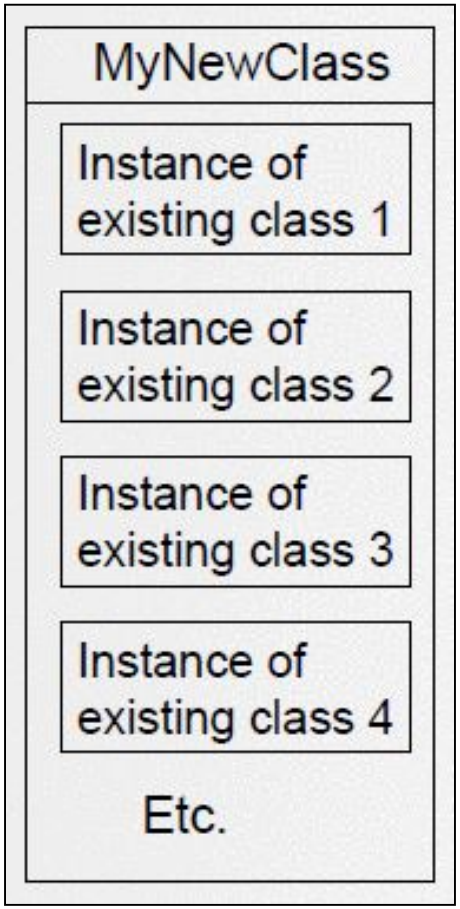
Reusing Classes

- When you need a class, you can:
  1) Get the perfect one off the shelf (one extreme)
  2) Write it completely from scratch (the other extreme)
  3) Reuse an existing class with composition
  4) Reuse an existing class or class framework with inheritance

Composition vs. Inheritance



Composition Syntax




```
class MyNewClass {
        Foo x = new Foo();
        Bar y = new Bar();
        Baz z = new Baz();
        // ...
}
```

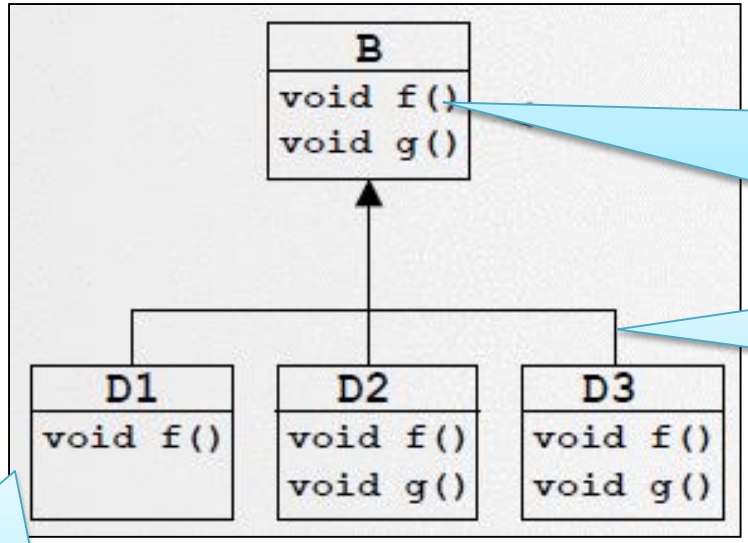
- Can also initialize in the constructor
- Flexibility: Can change objects at run time!
- "Has-A" relationship

Composition Syntax

```
class Soap {
  private String s;
  Soap() {
    System.out.println("Soap()");
    s = new String("Constructed");
  }
  public String toString() { return s; }
}

public class Bath {
  private String
    // Initializing at point of definition:
    s1 = new String("Happy"),
    s2 = "Happy",
    s3, s4;
  Soap castille;
  int i;
  float toy;
  Bath() {
    System.out.println("Inside Bath()");
    s3 = new String("Joy");
    i = 47;
    toy = 3.14f;
    castille = new Soap();
  }
  void print() {
    // Delayed initialization:
    if(s4 == null)
      s4 = new String("Joy");
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
    System.out.println("s4 = " + s4);
    System.out.println("i = " + i);
    System.out.println("toy = " + toy);
    System.out.println("castille = " + castille);
  }
  public static void main(String[] args) {
    Bath b = new Bath();
    b.print();
  }
} ///:~
```

Inheritance Syntax

```
class B {
        public void f(){
        /* … */ }
        public void g(){
        /* … */ }
}
```



Base interface automatically duplicated in derived classes

Base data members also duplicated

If a derived member is not redefined, base definition is used

```
class D1 extends B {
public void f() {
        /* … */
        }
}
```

Inheritance Syntax

```
class Cleanser {
  private String s = new
String("Cleanser");
  public void append(String a) { s +=
a; }
  public void dilute() { append("
dilute()"); }
  public void apply() { append("
apply()"); }
  public void scrub() { append("
scrub()"); }
  public void print()
{ System.out.println(s); }
  public static void main(String[] args) {
    Cleanser x = new Cleanser();
    x.dilute(); x.apply(); x.scrub();
    x.print();
  }
}
```

```
public class Detergent extends Cleanser {
  // Change a method:
  public void scrub() {
    append(" Detergent.scrub()");
    super.scrub(); // Call base-class version
  }
  // Add methods to the interface:
  public void foam() { append(" foam()"); }
  // Test the new class:
  public static void main(String[] args) {
    Detergent x = new Detergent();
    x.dilute();
    x.apply();
    x.scrub();
    x.foam();
    x.print();
    System.out.println("Testing base class:");
    Cleanser.main(args);
  }
} ///:~
```

Initializing the Base class

- Java automatically calls default constructors

```
class Art {
  Art() {
    System.out.println("Art constructor");
  }
}

class Drawing extends Art {
  Drawing() {
    System.out.println("Drawing constructor");
  }
}
public class Cartoon extends Drawing {
  Cartoon() {
    System.out.println("Cartoon constructor");
  }
  public static void main(String[] args) {
    Cartoon x = new Cartoon();
  }
} ///:~
```

>>

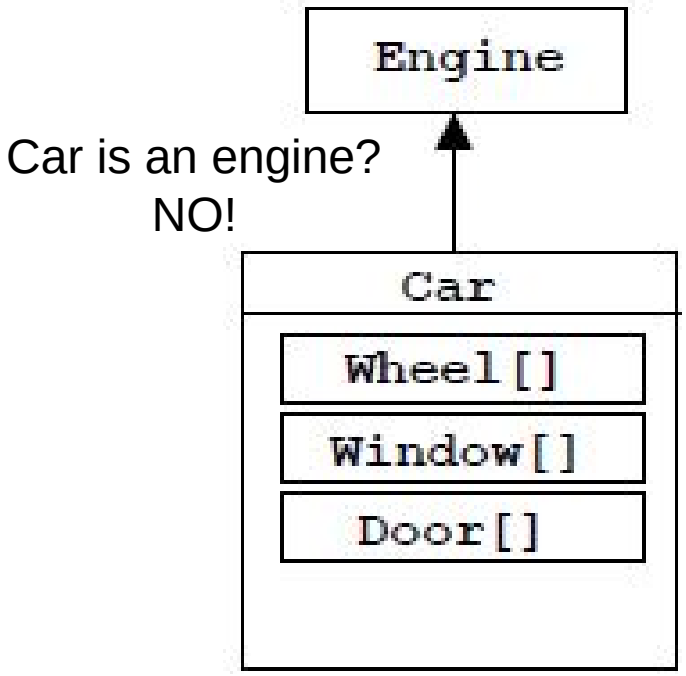
Art constructor
Drawing constructor
Cartoon constructor

Constructors with Arguments

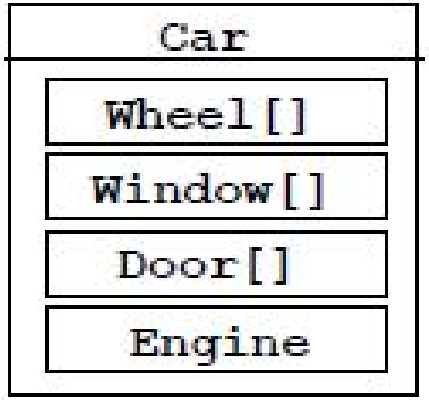
```
class Game {
  Game(int i) {
    System.out.println("Game constructor");
  }
}

class BoardGame extends Game {
  BoardGame(int i) {
    super(i);
    System.out.println("BoardGame constructor");
  }
}

public class Chess extends BoardGame {
  Chess() {
    super(11);
    System.out.println("Chess constructor");
  }
  public static void main(String[] args) {
    Chess x = new Chess();
  }
} ///:~
```

- Base constructor call must happen first
- Use **super** keyword

Choosing Composition vs. Inheritance



WRONG



RIGHT

- Inheritance is determined at compile time; member binding can be delayed until run time
- Member rebinding is possible
- In general, prefer composition to inheritance as a first choice

Initialization with Inheritance

```
class Insect {
  int i = 9;
  int j;
  Insect() {
    prt("i = " + i + ", j = " + j);
    j = 39;
  }
  static int x1 =
    prt("static Insect.x1 initialized");
  static int prt(String s) {
    System.out.println(s);
    return 47;
  }
}
public class Beetle extends Insect {
  int k = prt("Beetle.k initialized");
  Beetle() {
    prt("k = " + k);
    prt("j = " + j);
  }
  static int x2 =
    prt("static Beetle.x2 initialized");
  public static void main(String[] args) {
    prt("Beetle constructor");
    Beetle b = new Beetle();
  }
} ///:~
```

>>
static Insect.x1 initialized
static Beetle.x2 initialized
Beetle constructor
i = 9, j = 0
Beetle.k initialized
k = 47
j = 39

The final Keyword

- Slightly different meanings depending on context
- "This cannot be changed"
- A bit confusing: two reasons for using it
  - Design
  - Efficiency
- **final** fields
- **final** methods
- **final** class

final Fields

1) Compile-time constant

Must be given a value at point of definition

final static int NINE = 9;

May be "folded" into a calculation by the compiler

2) Run-time constant

Cannot be changed from initialization value

final int RNUM = (int)(Math.random()*20);

- **final static**: only one instance per class, initialized at the time the class is loaded, cannot be changed.
- **final** references: cannot be re-bound to other objects

final arguments & final Classes

- **Final arguments**
  - **void f(final int i) { // …**
    - Primitives can't be changed inside method
  - **void g(final Bob b) { // …**
    - References can't be rebound inside method
    - Generally, neither one is used
- **Final Class**
  - Cannot inherit from **final** class
  - All methods are implicitly **final**
  - Fields may be **final** or not, as you choose

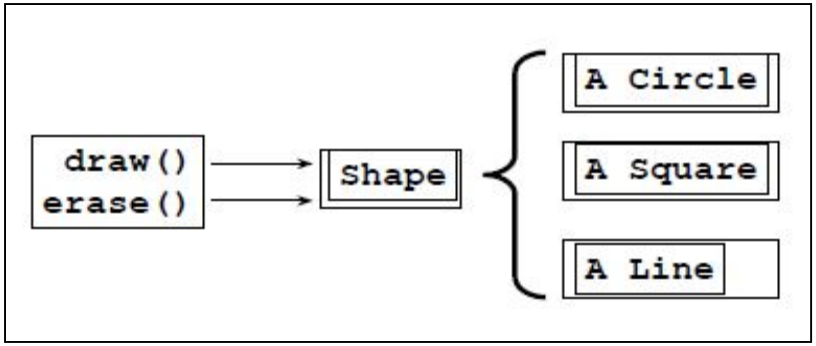
final Methods

1) Put a "lock" on a method to prevent any inheriting class from overriding it (design)

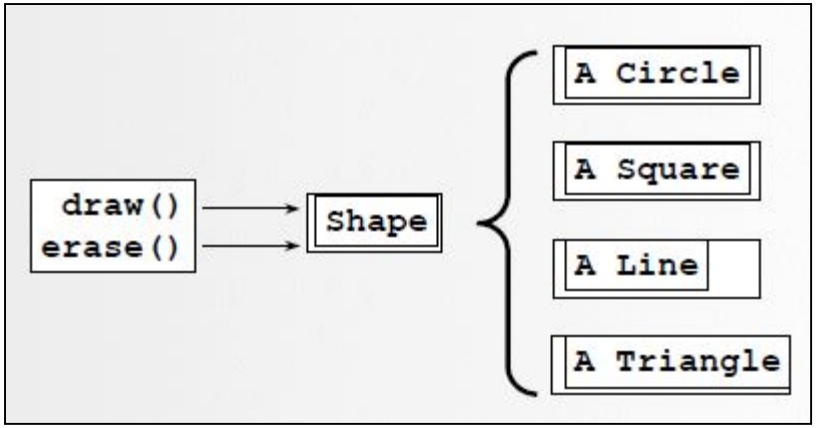
2) Efficiency (try to avoid the temptation…)
   - Compiler has permission to "inline" a final method
     - Replace method call with code
     - Eliminate method-call overhead
   - Programmers are characteristically bad about guessing where performance problems are
   - Limits use of class (example: can't override Vector)

- Private methods are implicitly final

Polymorphism



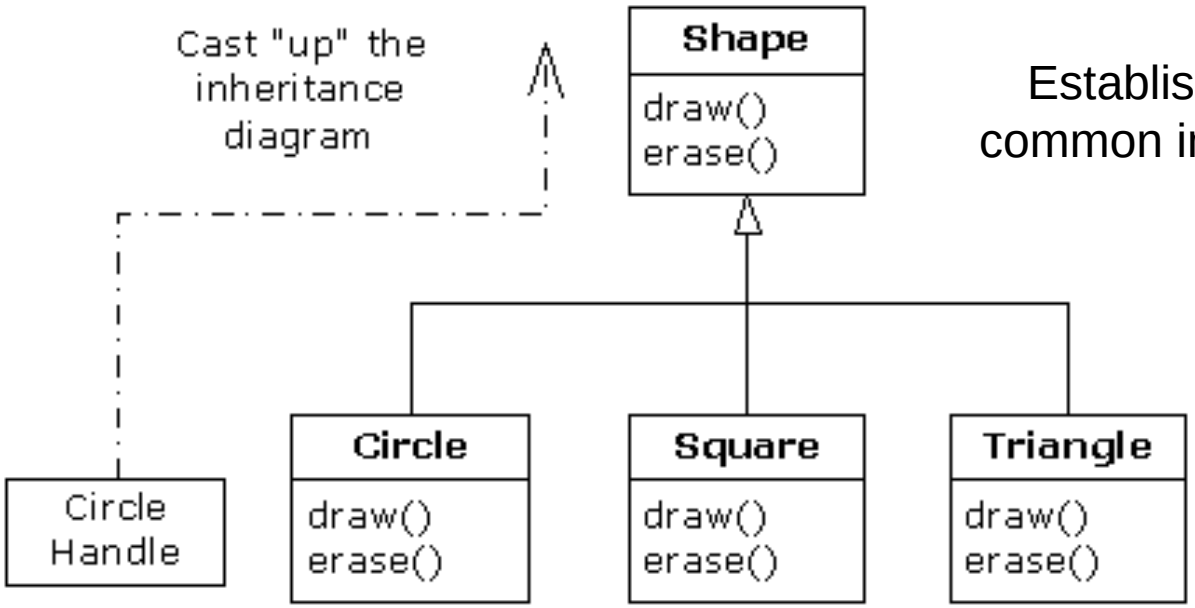
"Substitutability"



"Extensibility"

Interface & Implementation

Shape s = new Circle();



Establishes a common interface

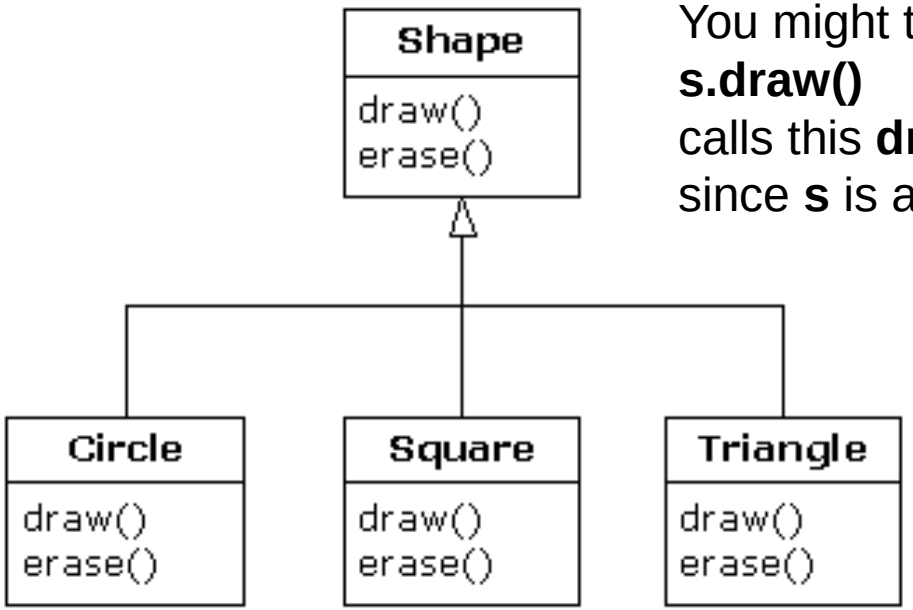
Implementations of the common interface

Why Upcast?

- All true OOP programs have some upcasting somewhere
- Upcasting allows us to isolate type specific details from the bulk of your code, i.e. decoupling
- Code is simpler to write and read
- Most important
  - Changes in type do not propagate changes in code

A problem

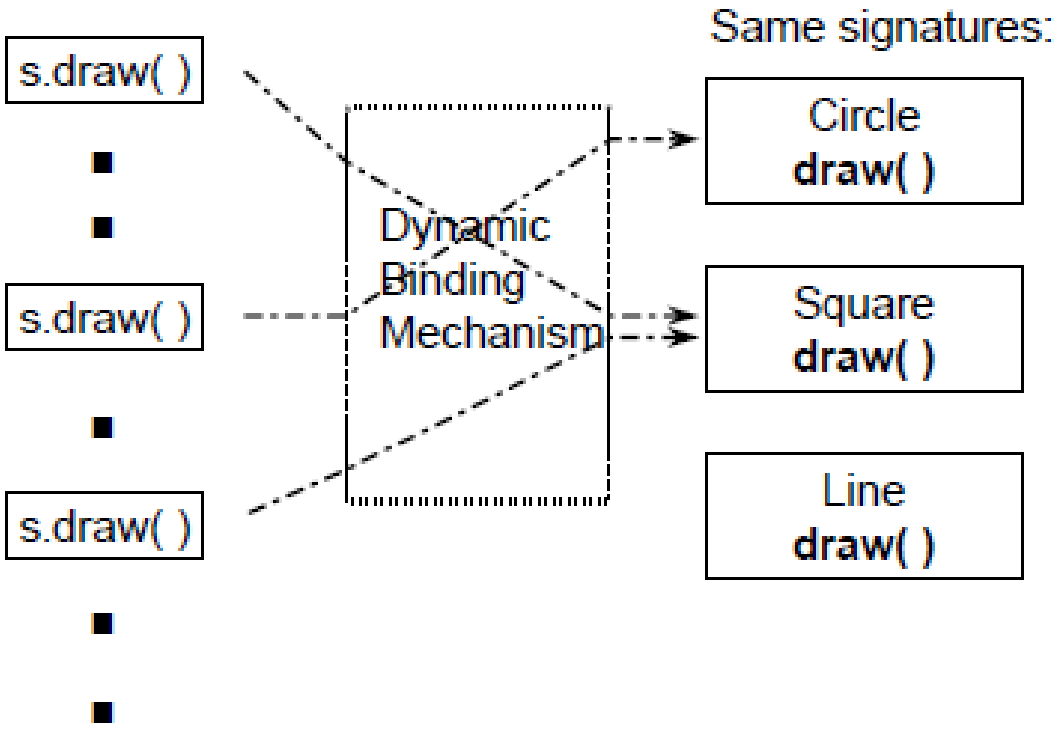
Shape s = new Circle();



You might think **s.draw()** calls this **draw()** since **s** is a **Shape**

We want it to call this **draw()** since **s** actually points to a **Circle**

Dynamic Binding in Java



Dynamic Binding in Java

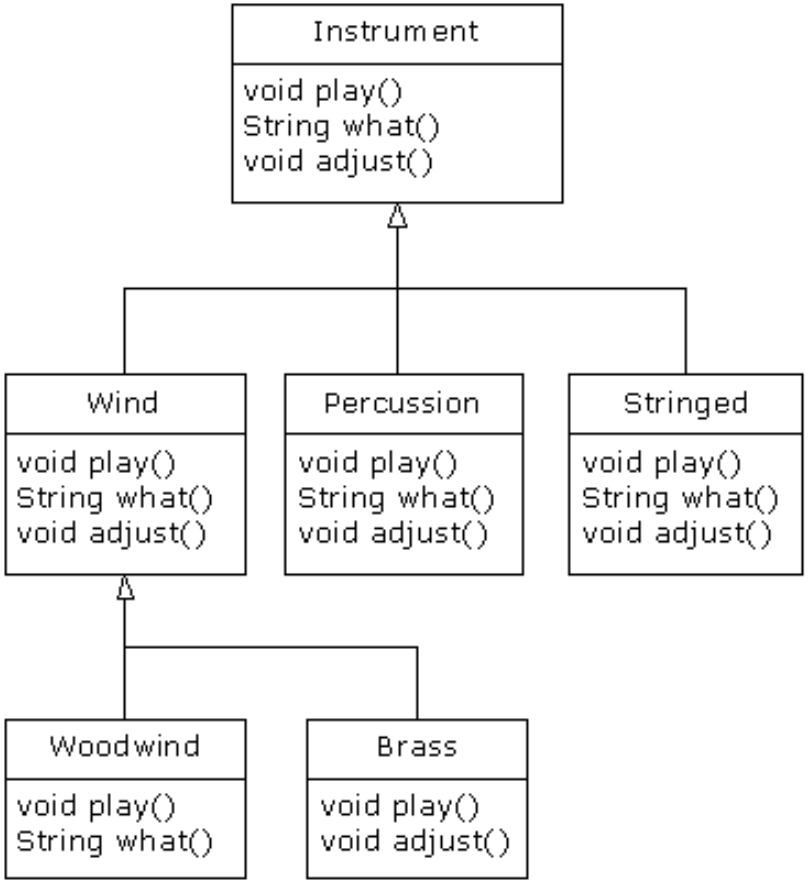
```
class Shape {
  void draw() {}
  void erase() {}
}

class Circle extends Shape {
  void draw() {
    System.out.println("Circle.draw()"); }
  }
  void erase() {
    System.out.println("Circle.erase()");
  }
}

class Square extends Shape {
  void draw() {
    System.out.println("Square.draw()");
  }
  void erase() {
    System.out.println("Square.erase()");
  }
}
```

```
class Triangle extends Shape {
  void draw() {
    System.out.println("Triangle.draw()");
  }
  void erase() {
    System.out.println("Triangle.erase()");
  }

public class Shapes {
  public static Shape randShape() {
    switch((int)(Math.random() * 3)) {
      default:
      case 0: return new Circle();
      case 1: return new Square();
      case 2: return new Triangle();
    }
  }
```

```
public static void main(String[] args) {
    Shape[] s = new Shape[9];
    // Fill up the array with shapes:
    for(int i = 0; i < s.length; i++)
      s[i] = randShape();
    // Make polymorphic method calls:
    for(int i = 0; i < s.length; i++)
      s[i].draw();
  }
} ///:~
```

```
>>
Circle.draw()
Triangle.draw()
Circle.draw()
Circle.draw()
Circle.draw()
Square.draw()
Triangle.draw()
Square.draw()
Square.draw()
```

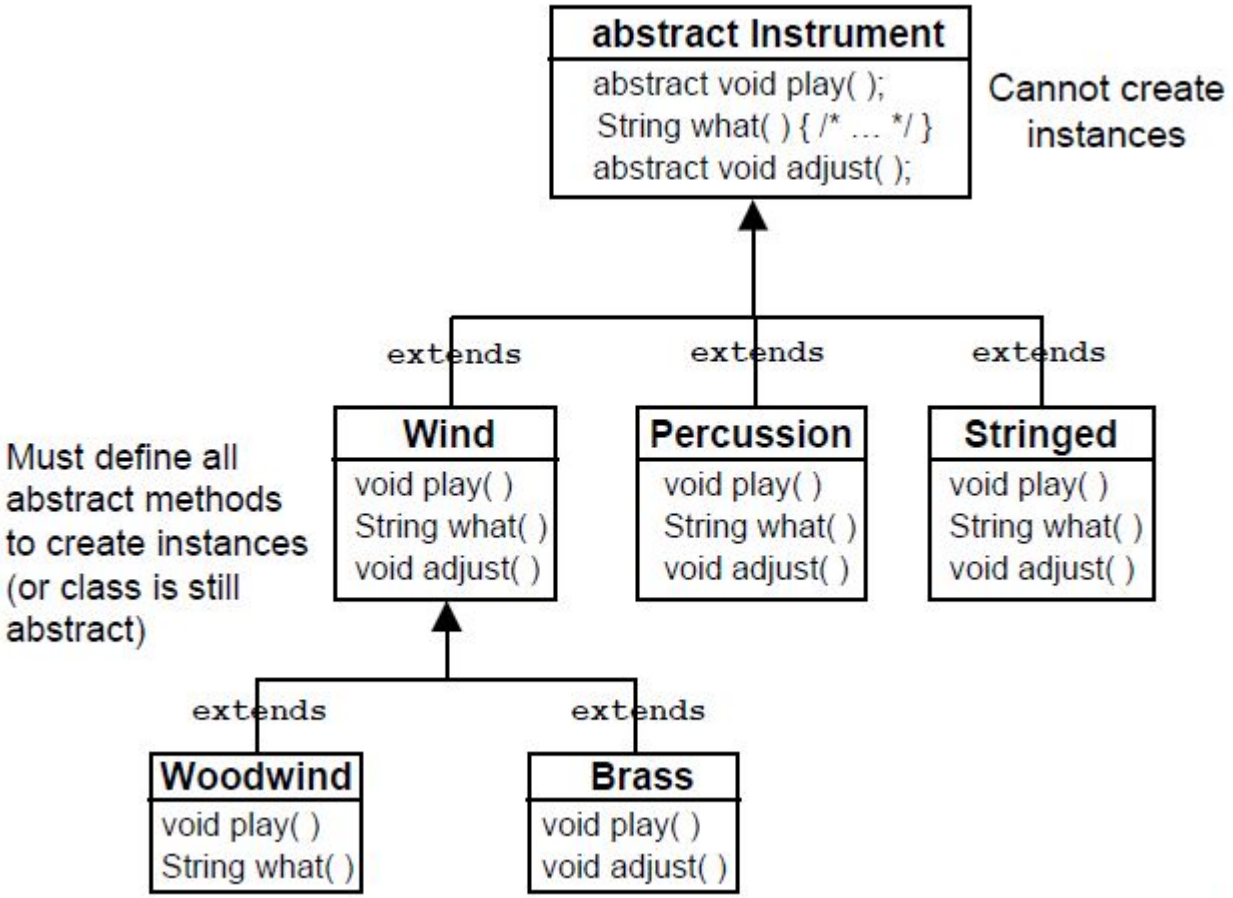
Extensibility



```
import java.util.*;

class Instrument {
  public void play() {
    System.out.println("Instrument.play()");
  }
  public String what() {
    return "Instrument";
  }
  public void adjust() {}
}

class Wind extends Instrument {
  public void play() {
    System.out.println("Wind.play()");
  }
  public String what() { return "Wind"; }
  public void adjust() {}
}

class Percussion extends Instrument {
  public void play() {
    System.out.println("Percussion.play()");
  }
  public String what() { return "Percussion"; }
  public void adjust() {}
}

class Stringed extends Instrument {
  public void play() {
    System.out.println("Stringed.play()");
  }
  public String what() { return "Stringed"; }
  public void adjust() {}
}

class Brass extends Wind {
  public void play() {
    System.out.println("Brass.play()");
  }
  public void adjust() {
    System.out.println("Brass.adjust()");
  }
}

class Woodwind extends Wind {
  public void play() {
    System.out.println("Woodwind.play()");
  }
  public String what() { return "Woodwind"; }
}

public class Music3 {
  // Doesn't care about type, so new types
  // added to the system still work right:
  static void tune(Instrument i) {
    // ...
    i.play();
  }
  static void tuneAll(Instrument[] e) {
    for(int i = 0; i < e.length; i++)
      tune(e[i]);
  }
  public static void main(String[] args) {
    Instrument[] orchestra = new Instrument[5];
    int i = 0;
    // Upcasting during addition to the array:
    orchestra[i++] = new Wind();
    orchestra[i++] = new Percussion();
    orchestra[i++] = new Stringed();
    orchestra[i++] = new Brass();
    orchestra[i++] = new Woodwind();
    tuneAll(orchestra);
  }
} ///:~
```

```
>>
Wind.play()
Percussion.play()
Stringed.play()
Brass.play()
Woodwind.play()
```

Abstract classes & abstract methods



1

An abstract Instrument

- Some methods and data may be defined

```
abstract class Instrument {
        int i; // storage allocated for each
        public abstract void play();
        public String what() {
                return "Instrument";
        }
        public abstract void adjust();
}
```

- Rest of the code is the same…
- This maps with C++, may have been an early design

Constructors & Polymorphism

Order of constructor calls

```
class Meal {
  Meal() { System.out.println("Meal()"); }
}

class Bread {
  Bread() { System.out.println("Bread()"); }
}

class Cheese {
  Cheese() { System.out.println("Cheese()"); }
}

class Lettuce {
  Lettuce() { System.out.println("Lettuce()"); }
}
```

```
class Lunch extends Meal {
  Lunch() { System.out.println("Lunch()");}
}

class PortableLunch extends Lunch {
  PortableLunch() {
    System.out.println("PortableLunch()");
  }
}

public class Sandwich extends PortableLunch {
  Bread b = new Bread();
  Cheese c = new Cheese();
  Lettuce l = new Lettuce();
  Sandwich() {
    System.out.println("Sandwich()");
  }
  public static void main(String[] args
    new Sandwich();
  }
} ///:~
```

```
> >
Meal()
Lunch()
PortableLunch()
Bread()
Cheese()
Lettuce()
Sandwich()
```

Order of Initialization

1) Base-class constructor is called

   This step is repeated recursively such that the very root of the hierarchy is constructed first, followed by the next derived class, etc., until the most derived class is reached

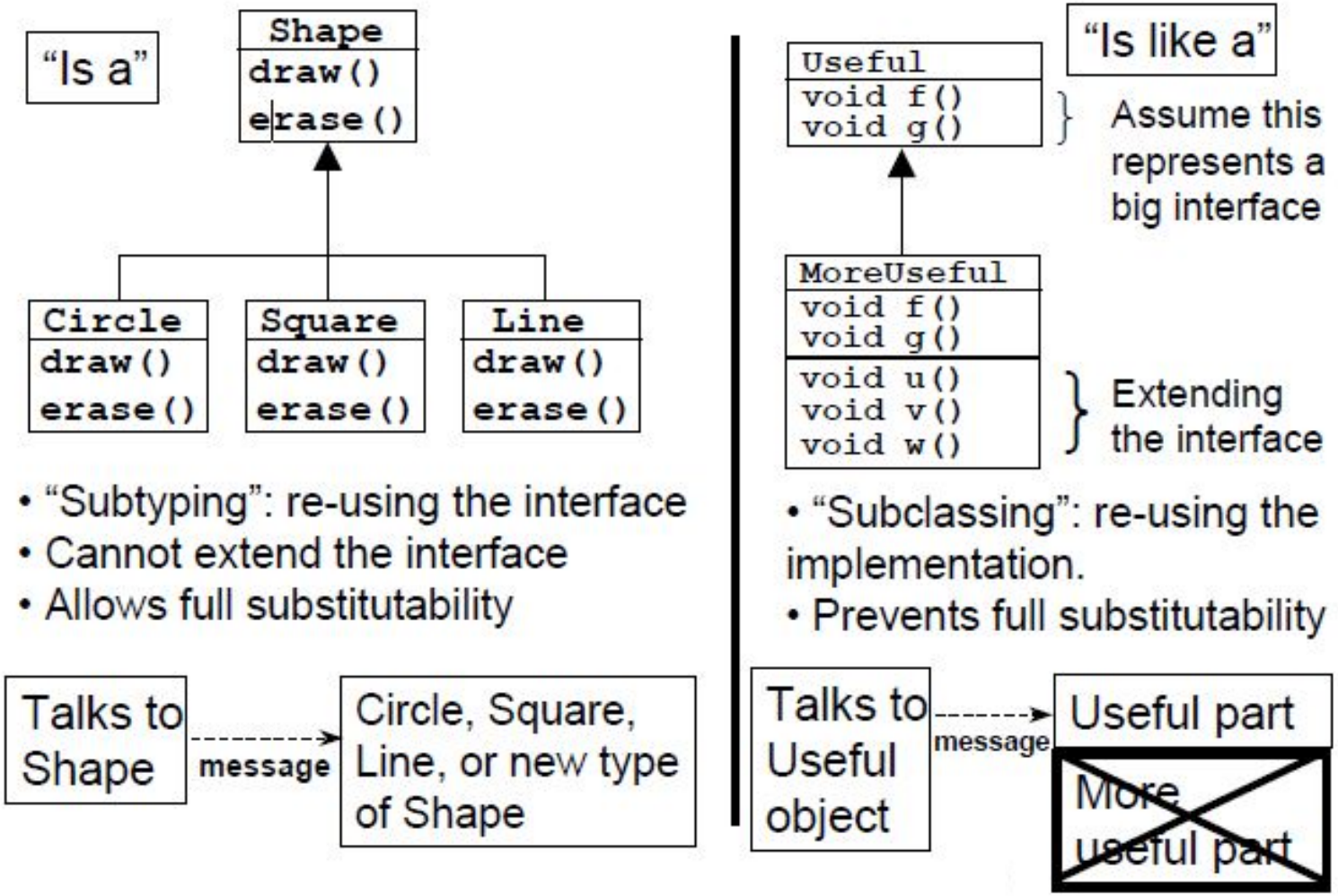
2) Member initializers are called in the order of declaration

3) Body of the derived-class constructor is called

Polymorphism & Constructors

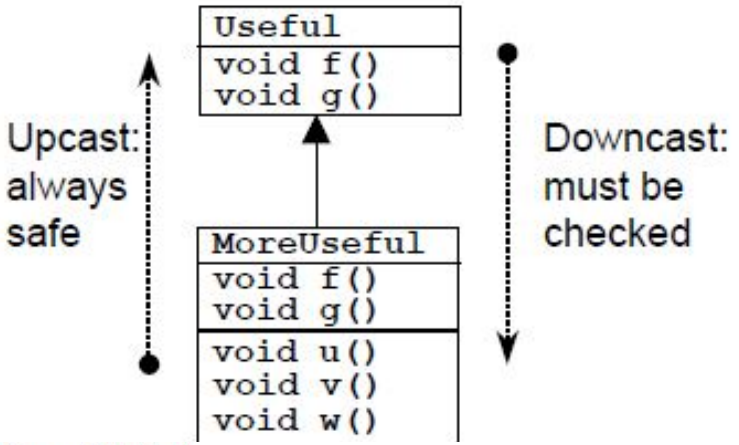
- Inside constructor
  - Overridden method *is* used!
- This can produce incorrect results
  - Overridden method assumes all parts of derived class have been initialized
  - But you're still in the base part of the constructor
  - Derived constructor hasn't been called yet!

Pure Inheritance vs. Extension



Downcasting & Run-Time Type Identification (RTTI)

- Normally try to do everything with upcasting
- If you extend the class, you must downcast to access extended methods
- Java casts are always checked at run-time (safe)




```
class Useful {
  public void f() {}
  public void g() {}
}

class MoreUseful extends Useful {
  public void f() {}
  public void g() {}
  public void u() {}
  public void v() {}
  public void w() {}
}
```

Extends the interface

```
public class RTTI {
  public static void main(String args[]) {
    Useful x[] = {
      new Useful(),
      new MoreUseful()
    };
    x[0].f();
    x[1].g();
    // Compile-time: method not found in Useful:
    //! x[1].u();
    ((MoreUseful)x[1]).u(); // Downcast/RTTI
    ((MoreUseful)x[0]).u(); // Exception thrown
  }
}
```

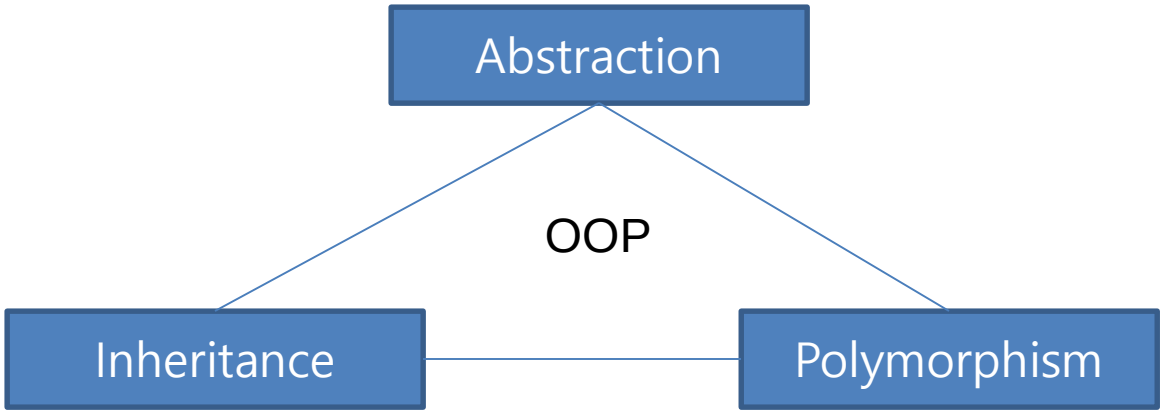
Summary

- Access control
  - What users can & can't use (shows the area of interest)
  - Separating interface & implementation
  - Allowing the class creator to change the implementation later without disturbing client code
  - An important design & implementation flexibility
- Design guideline
  - Always make elements "as **private** as possible"

Summary Cont'd

- Easy to think that OOP is *only* about inheritance
- Often easier and more flexible to start with composition
  - Remember to say "has-a" and "is-a"
- Use inheritance when it's clear that a new type is a kind of a base type

Summary Cont'd



- Not just creating types, but proper behavior in all situations
- Allows decoupling of code from specific type it's acting on: easy reading, writing & extensibility