Inheritance and Interfaces

Object Orientated Programming in Java

Benjamin Kenwright

Outline

- Review
- What is Inheritance?
- Why we need Inheritance?
  - Syntax, Formatting, ..
- What is an Interface?
- Today's Practical
- Review/Discussion

Inheritance

- Reuse
- Inheritance and methods
- Method redefinition
- The final keyword
- Comparison of inheritance with other approaches and examples

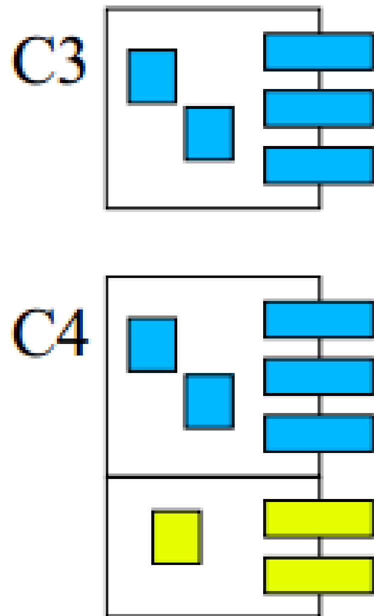
How to Reuse Code?

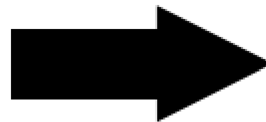
- Write the class completely from scratch (one extreme)
  - ▷ What some programmers always want to do!
- Find an existing class that exactly match your requirements (another extreme)
  - ▷ The easiest for the programmer!
- Built it from well-tested, well-documented existing classes
  - ▷ A very typical reuse, called composition reuse!
- Reuse an existing class with inheritance
  - ▷ Requires more knowledge than composition reuse

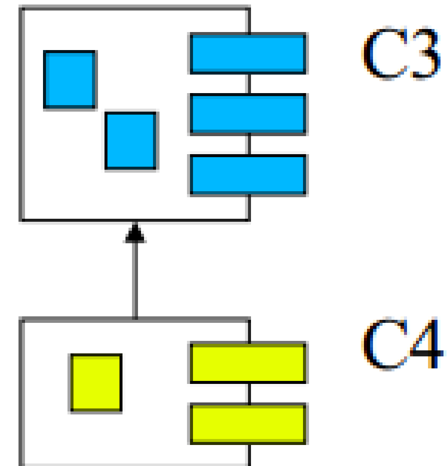
Inheritance

- Inheritance is a way to derive a new class from an existing class
- It is the process where one object acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order

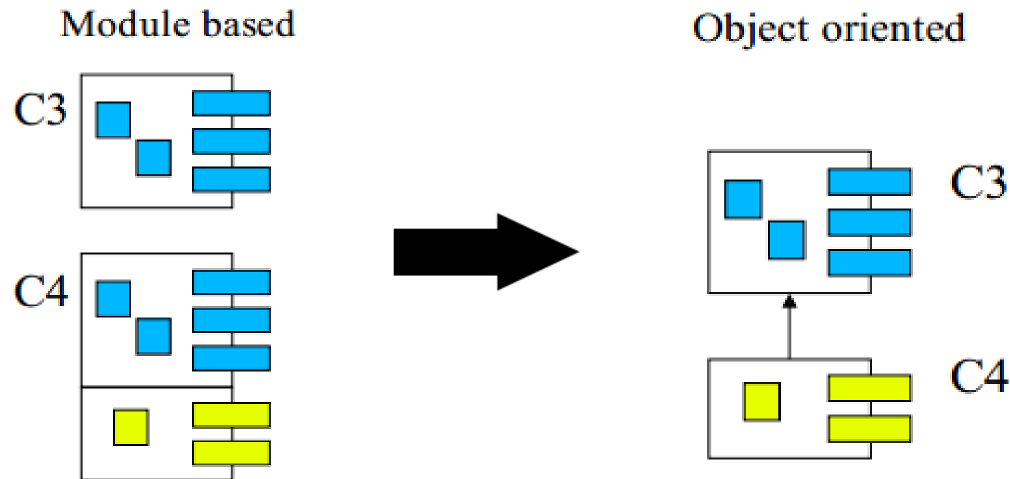
Module Based vs. Object Oriented







Module Based vs. Object Oriented



- Class C4 is created by copying C3
- There are C3 and C4 instances
- Instance of C4 have all C3 properties
- C3 and C4 are totally separated
- Maintenance of C3 properties must be done two places
- Languages, e.g., Ada, Modula2, PL/SQL

- Class C4 inherits from C3
- There are C3 and C4 instances
- Instance of C4 have all C3 properties
- C3 and C4 are closely related
- Maintenance of C3 properties must be done in one place.
- Languages, C++, C#, Java, Smalltalk

Question

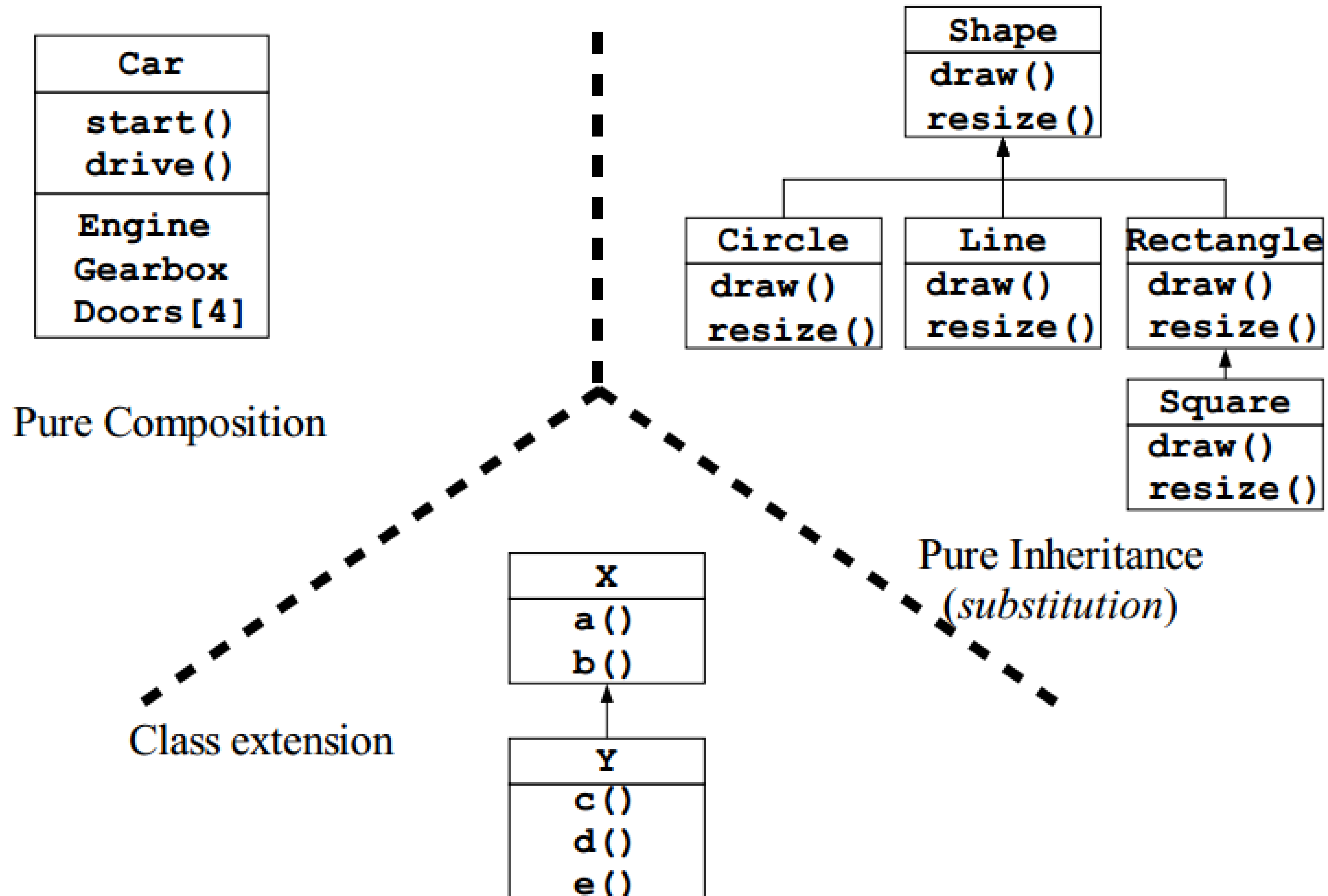
- Inheritance is a way to derive a new class from an existing class

a) True
b) False

Answer

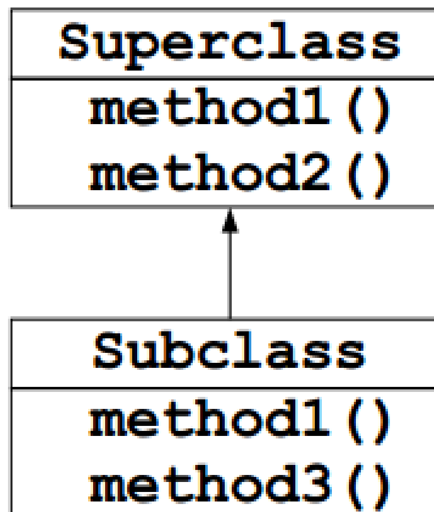
- a) True

Composition vs. Inheritance



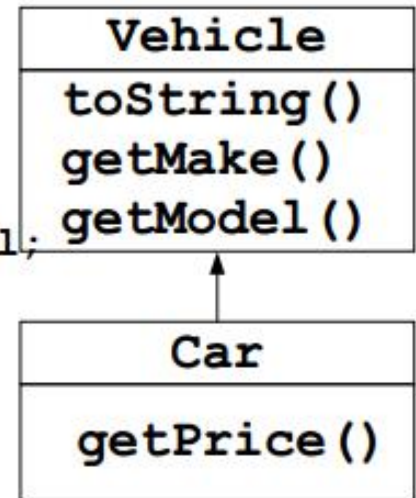
Inheritance in Java (Syntax)

```
class Subclass extends Superclass {
    // <class body>
}
```



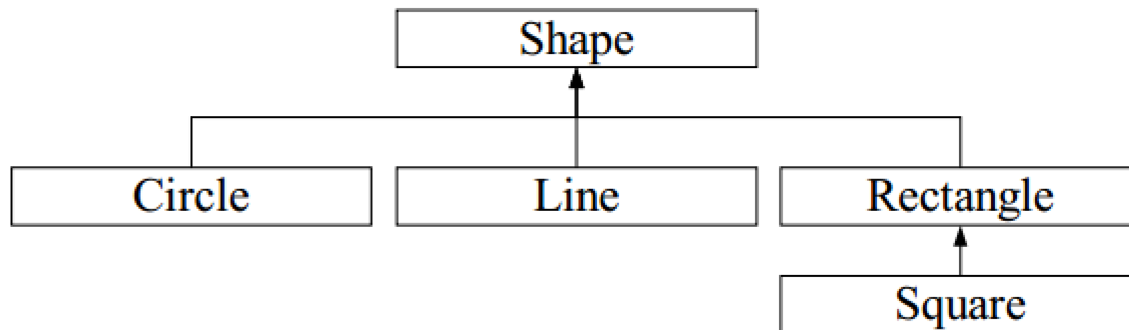
Inheritance Example

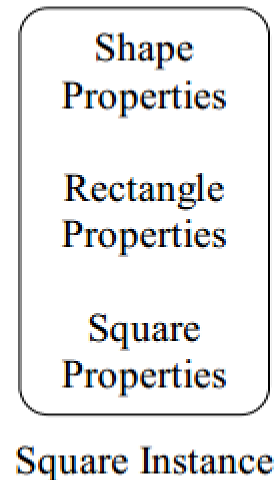
```
public class Vehicle {
    private String make;
    private String model;
    public Vehicle() { make = ""; model = "";}
     public String toString() {
         return "Make: " + make + " Model: " + model;
     }
    public String getMake(){ return make; }
    public String getModel() { return model; }
}
public class Car extends Vehicle {
    private double price;
    public Car() {
        super(); // called implicitly can be left outs
        price = 0.0;
    }
    public String toString() { // method overwrites
        return "Make: " + getMake() + " Model: " + getModel()
                        + " Price: " + price;
    }
    public double getPrice(){ return price; }
}
```



Instantiating and Initialization

- The Square, that inherits from Rectangle, that inherits from Shape is instantiated as a single object, with properties from the three classes Square, Rectangle, and Shape





Square Instance

Inheritance and Constructors

- Constructors are not inherited
- A constructor in a subclass must initialize variables in the class and variables in the superclass
  - What about private fields in the superclass?
- It is possible to call the superclass' constructor in a subclass
  - Default superclass constructor called if exists

```
public class Vehicle{
    private String make, model;
    public Vehicle(String ma, String mo) {
        make = ma; model = mo;
    }
}
public class Car extends Vehicle{
    private double price;
    public Car() {
        // System.out.println("Start");  // not allowed
        super("", "");                   // must be called
        price = 0.0;
    }
}
```

Order of Instantiation and Initialization

- The storage allocated for the object is initialized to binary zero before anything else happens
- Static initialization in the base class then the derived classes
- The base-class constructor is called (all the way up to Object)
- Member initializers are called in the order of declaration
- The body of the derived-class constructor is called

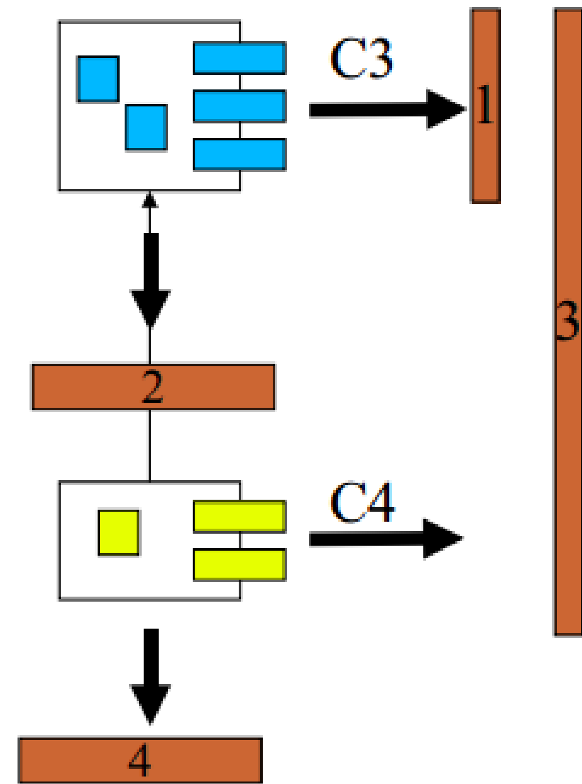
Inheritance and Constructors, cont.

```
class A {
    public A(){
        System.out.println("A()");
        // when called from B the B.doStuff() is called
        doStuff();
    }
    public void doStuff(){System.out.println("A.doStuff()"); }
}
class B extends A{
    int i = 7;
    public B(){System.out.println("B()");}
    public void doStuff(){System.out.println("B.doStuff() " + i);
    }

}

public class Base{
    public static void main(String[] args){
        B b = new B();
        b.doStuff();
    }
}
```

//prints
A()
B.doStuff() 0
B()
B.doStuff() 7

Interface to Subclasses and Clients

1. The properties of C3 that clients can use
2. The properties of C3 that C4 can use
3. The properties of C4 that clients can use
4. The properties of C4 that subclasses of C4 can use



Question

- All methods are inherited including the constructors?

a) True

b) False

Answer

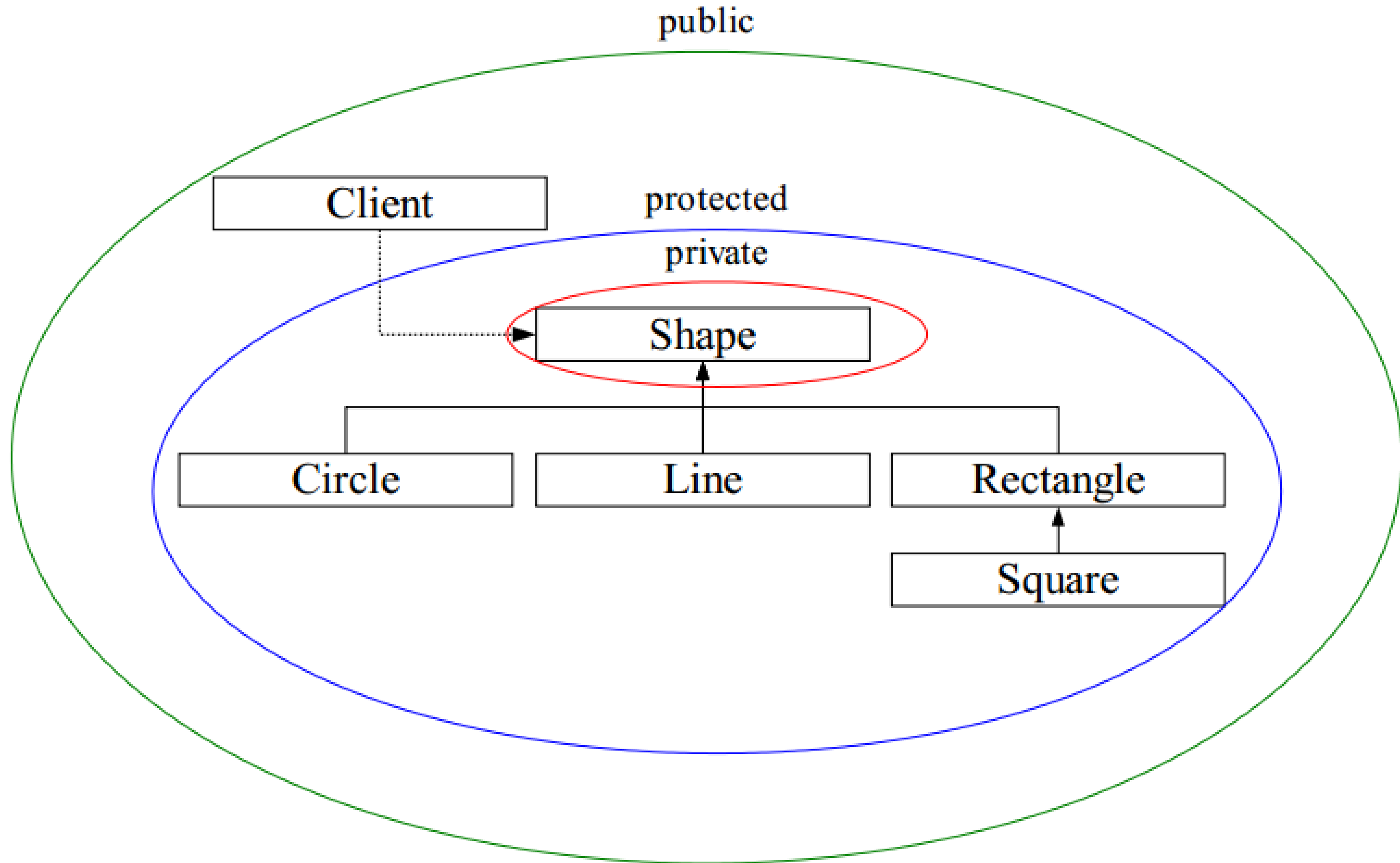
- b) False

Constructors are not inherited

“Protected” Revisited

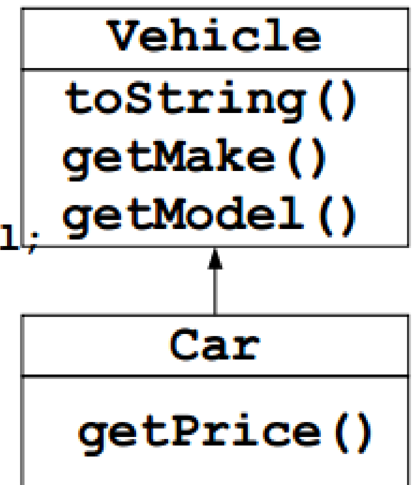
- It must be possible for a subclass to access properties in a superclass
  - private will not do, it is to restrictive
  - public will not do, it is to generous
- A protected variable or method in a class can be accessed by subclasses but not by clients
- Change access modifiers when inheriting
  - Properties can be made “more public”
  - Properties cannot be made “more private”

“Protected” Revisited



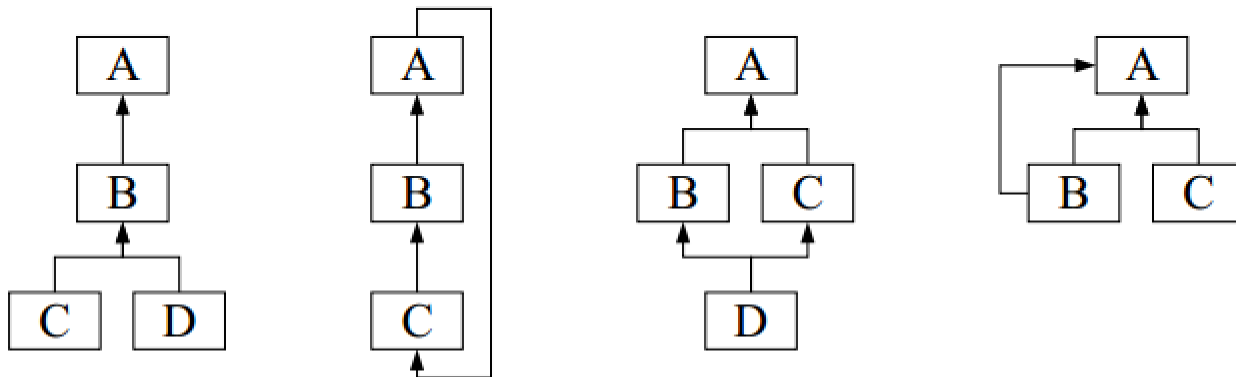
“Protected” Example

```
public class Vehicle1 {
    protected String make;
    protected String model;
    public Vehicle1() {    make = ""; model = "";}
     public String toString() {
          return "Make: " + make + " Model: " + model;
     }
    public String getMake(){ return make;}
    public String getModel() { return model;}
}

public class Car1 extends Vehicle1 {
    private double price;
    public Car1() {
        price = 0.0;
    }
    public String toString() {
        return "Make: " + make + " Model: " + model
                        + " Price: " + price;
    }
    public double getPrice(){ return price; }
}
```



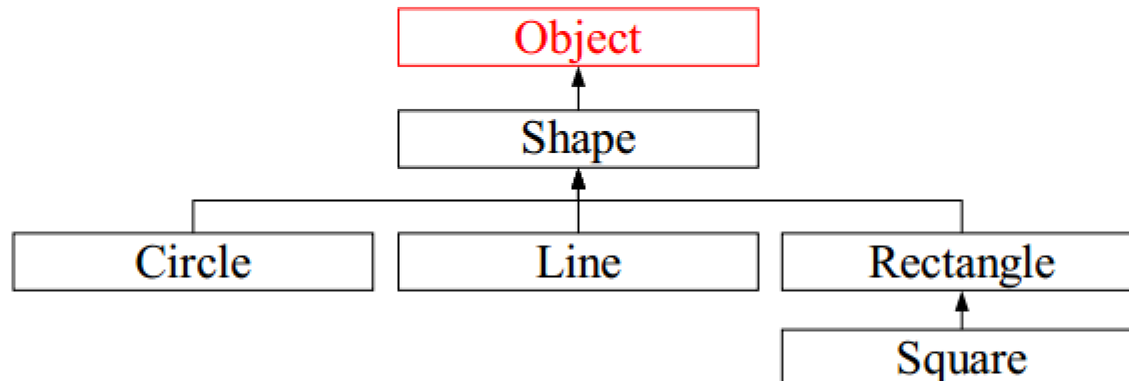
Class Hierarchies in General

- Class hierarchy: a set of classes related by inheritance
- Possibilities with inheritance
  - Cycles in the inheritance hierarchy is not allowed
  - Inheritance from multiple superclass may be allowed
  - Inheritance from the same superclass more than once may be allowed



Class Hierarchies in Java

- Class Object is the root of the inheritance hierarchy in Java
- If no superclass is specified a class inherits implicitly from Object
- If a superclass is specified explicitly the subclass will inherit Object



Question

- A protected variable or method in a class cannot be accessed by subclasses but not by clients

a) True

b) False

Answer

- False

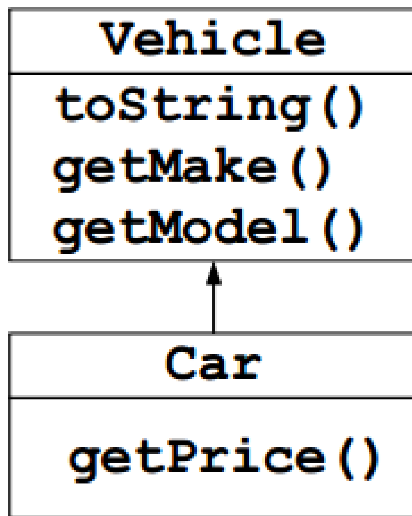
A protected variable or method in a class **can** be accessed by subclasses but not by clients

Method/Variable Redefinition

- Redefinition: A method/variable in a subclass has the same name as a method/variable in the superclass
- Redefinition should change the implementation of a method, not its semantics
- Redefinition in Java class B inherits from class A if
  - Method: Both versions of the method is available in instances of B. Can be accessed in B via super
  - Variable: Both versions of the variable is available in instances of B. Can be accessed in B via super

Upcasting

- Treat a subclass as its superclass



```
// example
Car c = new Car();
Vehicle v;
v = c;              // upcast

v.toString();    // okay
v.getMake();     // okay
//v.getPrice(); // not okay
```

- Central feature in object-oriented program

The final Keyword

- Fields
  - ▷Compile time constant (very useful)

    `final static double PI = 3.14`
  - ▷Run-time constant (useful)

    `final int RAND = (int) Math.random * 10`
- Arguments (not very useful)

  `double foo (final int i)`
- Methods
  - ▷Prevents overwriting in a subclass (use this very carefully)
  - ▷Private methods are implicitly final
- Final class (use this very carefully)
  - ▷Cannot inherit from the class

Question

- Which of these keywords can be used to prevent inheritance of a class?

a) super

b) constant

c) Class

d) final

Answer

- d) final

Review

- Reuse
  - Use composition when ever possible more flexible and easier to understand than inheritance
- Java supports specialization and extension via inheritance
  - Specialization and extension can be combined.
- A subclass automatically gets the fields and method from the superclass
  - They can be redefined in the subclass
- Java supports single inheritance, all have Object as superclass
- Designing good reusable classes is (very) hard!

```
while(!goodDesign()){ reiterateTheDesign(); }
```

Question

- Which of these classes is a superclass of every class in Java?

a) String class

b) Object class

c) Abstract class

d) ArrayList class

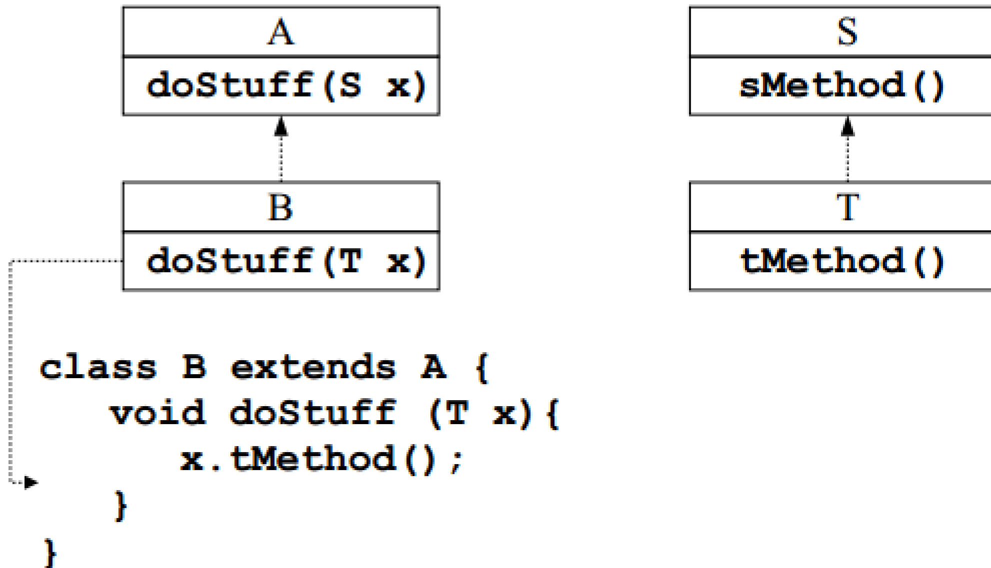
Answer

- b) Object class

Method Combination

- It is programmatically controlled
  - Method doStuff on A controls the activation of doStuff on B
  - Method doStuff on B controls the activation of doStuff on A
  - Imperative method combination
- There is an overall framework in the run-time environment that controls the activation of doStuff on A and B
  - doStuff on A should not activate doStuff on B, and vice versa
  - Declarative method combination
- Java supports imperative method combination

Changing Parameter and Return Types




```
A a1 = new A();
B b1 = new B();
S s1 = new S();

a1 = b1;
a1.doStuff (s1);  // can we use an S object here?
```

Question

What would the following program output?

a) "sMethod“
b) “tMethod”
c) “doStuff(S x)”
d) “doStuff(T x)”
e) Nothing

```
class S {
 void sMethod() { System.out.print("sMethod"); }
}

class T extends S {
 void tMethod() { System.out.print("tMethod"); }
}

class A {
  void doStuff(S x) { System.out.print("doStuff(S x)");  }
}

class B extends A {
 void doStuff( T x ) {
        System.out.print("doStuff(T x)");
        x.tMethod();
 } }

class Test
{                   public static void main(String[] args)
                  {
                                 A a1 = new A();
                                 B b1 = new B();
                                 S s1 = new S();

                                 a1 = b1;
                                 a1.doStuff( s1 );
}   }
```

Answer

c) “doStuff(S x)”

Question

- Java supports multiple inheritance and all objects are superclasses

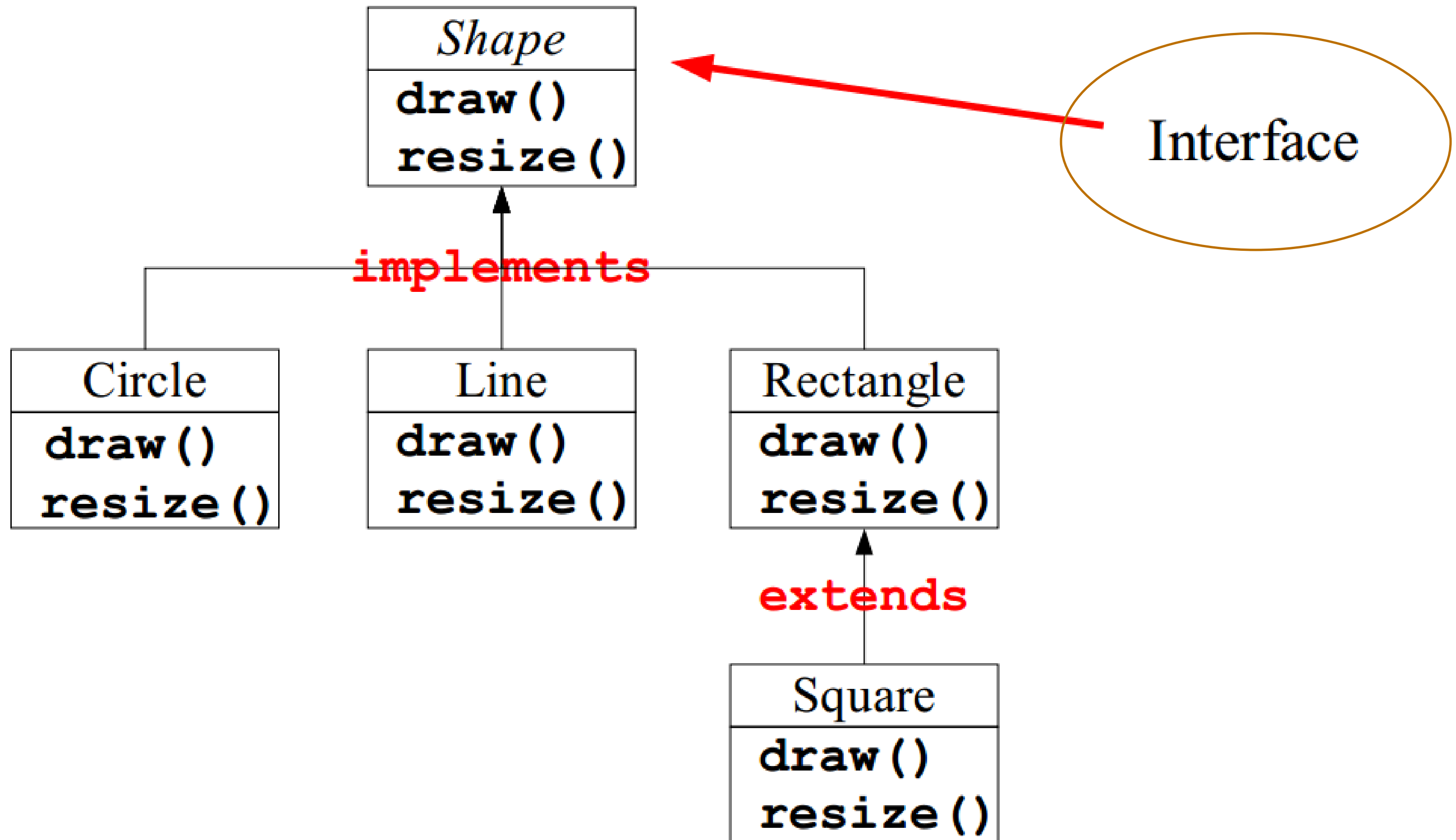
a) True
b) False

Answer

- b) False

Java supports single inheritance, all have Object as superclass

Java's interface Concept



Java's interface Concept, cont.

```
public interface Shape {
    double PI = 3.14;   // static and final => upper case
    void draw();        // automatic public
    void resize();      // automatic public
}

public class Rectangle implements Shape {
    public void draw() {System.out.println ("Rectangle"); }
    public void resize() { /* do stuff */ }

}

public class Square extends Rectangle {
    public void draw() {System.out.println ("Square"); }
    public void resize() { /* do stuff */ }

}
```

Java's interface Concept

- An interface is a collection of method declarations
  - An interface is a class-like concept
  - An interface has no variable declarations or method bodies
- Describes a set of methods that a class can be forced to implement
- An interface can be used to define a set of "constant"
- An interface can be used as a type concept.
  - Variable and parameter can be of interface types
- Interfaces can be used to implement multiple inheritance like hierarchies

Java's interface Concept

```
interface InterfaceName {
    // "constant" declarations
    // method declarations
}

// inheritance between interfaces
interface InterfaceName extends InterfaceName {
    ...
}

// not possible
interface InterfaceName extends InterfaceName1, InterfaceName2
{
    ...
}

// not possible
interface InterfaceName extends ClassName {  ... }
```

Java's interface Concept, cont

```
// implements instead of extends
class ClassName implements InterfaceName {
    ...
}

// multiple inheritance like
class ClassName implements InterfaceName1, InterfaceName2
{
    ...
}

// combine inheritance and interface implementation
class ClassName extends SuperClass implements InterfaceName
{
    ...
}

// multiple inheritance like again
class ClassName extends SuperClass
        implements InterfaceName1, InterfaceName2 {
    ...
}
```

Question

- An interface is allowed variable declarations but no method implementations

a) True

b) False

Answer

- b) False

interface has no variable declarations or method bodies

Multiple Inheritance

- Multiple inheritance of implementations is not allowed. Components can inherit multiple interfaces, though
- Inheriting multiple interfaces isn't problematic, since you're simply defining new method signatures to be implemented. It's the inheritance of multiple copies of functionality that is traditionally viewed as causing problems, or at the very least, confusion

Semantic Rules for Interfaces

- Type
  - An interface can be used as a type, like classes
  - A variable or parameter declared of an interface type is polymorph
    - Any object of a class that implements the interface can be referred by the variable
- Instantiation
  - Does not make sense on an interface
- Access modifiers
  - An interface can be public or "friendly" (the default)
  - All methods in an interface are default abstract and public
    - Static, final, private, and protected cannot be used.
  - All variables ("constants") are public static final by default
    - Private, protected cannot be used

Some of Java's Most used Interfaces

- *Iterator*
  - To run through a collection of objects without knowing how the objects are stored, e.g., in array, list, bag, or set.
  - More on this in the lecture on the Java collection library
- *Cloneable*
  - To make a copy of an existing object via the clone() method on the class Object
  - More on this topic in todays lecture
- *Serializable*
  - Pack a web of objects such that it can be send over a network or stored to disk. An naturally later be restored as a web of objects
  - More on this in the lecture on Java's I/O system
- *Comparable*
  - To make a total order on objects, e.g., 3, 56, 67, 879, 3422, 34234

The Iterator Interface

- The Iterator interface in the package java.util is a basic iterator that works on collections.

```
package java.util;
public interface Iterator {
    // the full meaning is public abstract boolean hasNext()
    boolean hasNext();
    Object next();
    void remove(); // optional throws exception
}
```

```
// use an iterator
myShapes = getSomeCollectionOfShapes();
Iterator iter = myShapes.iterator();
while (iter.hasNext()) {
  Shape s = (Shape)iter.next(); // downcast
  s.draw();
}
```

The Cloneable Interface

- A class X that implements the Cloneable interface tells clients that X objects can be cloned
- The interface is empty, i.e., has no methods
- Returns an identical copy of an object.
  - A shallow copy, by default.
  - A deep copy is often preferable.

The Cloneable Interface, Example

```
// Car example revisited
public class Car implements Cloneable{
    private String make;
    private String model;
    private double price;
   // default constructor
   public Car() {
        this("", "", 0.0);
    }
   // give reasonable values to instance variables
   public Car(String make, String model, double price){
        this.make  = make;
        this.model = model;
        this.price = price;
    }
    // the Cloneable interface
    public Object clone(){
        return new Car(this.make, this.model, this.price);
    }
}
```

The Serializable Interface

- A class X that implements the Serializable interface tells clients that X objects can be stored on file or other persistent media
- The interface is empty, i.e., has no methods

```
public class Car implements Serializable {
  // rest of class unaltered
  snip
}

// write to and read from disk
import java.io.*;
public class SerializeDemo{
    Car myToyota, anotherToyota;
    myToyota = new Car("Toyota", "Carina", 42312);
    ObjectOutputStream out = getOutput();
    out.writeObject(myToyota);

    ObjectInputStream in = getInput();
    anotherToyota = (Car)in.readObject();
}
```

The Comparable Interface

- In the package java.lang
- Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object

```
package java.lang;
public interface Comparable {
    int compareTo(Object o);
}
```

The Comparable Interface, Example

```
// IPAddress example revisited
public class IPAddress implements Comparable{
    private int[] n; // here IP stored, e.g., 125.255.231.123

    /** The Comparable interface */
    public int compareTo(Object o){
        IPAddress other = (IPAddress) o; // downcast
        int result = 0;
        for(int i = 0; i < n.length; i++){
            if (this.getNum(i) < other.getNum(i)){
                result = -1;
                break;
            }
            if (this.getNum(i) > other.getNum(i)){
                result = 1;
                break;
            }
        }
        return result;
    }
}
```

Interface vs. Abstract Class

Interface

- Methods can be declared
- No method bodies
- “Constants” can be declared
- Has no constructors
- Multiple inheritance possible
- Has no top interface
- Multiple “parent” interfaces

Abstract Class

- Methods can be declared
- Method bodies can be defined
- All types of variables can be declared
- Can have constructors
- Multiple inheritance not possible
- Always inherits from Object
- Only one “parent” class

Interfaces and Classes Combined

- Using interfaces objects do not reveal which classes the belong to
  - With an interface it is possible to send a message to an object without knowing which class(es) it belongs. The client only knows that certain methods are accessible
  - By implementing multiple interfaces it is possible for an object to change role during its life span.
- Design guidelines
  - Use classes for specialization and generalization
  - Use interfaces to add properties to classes

Review

- Purpose: Interfaces and abstract classes can be used for program design, not just program implementation
- Java only supports single inheritance
- Java “fakes” multiple inheritance via interfaces
  - Very flexible because the object interface is totally separated from the objects implementation

Summary

- Overview of Inheritance and Interfaces
- Hands-On/Practical to help gain a solid understanding of these concepts
- Today is about becoming comfortable/familiar these core Object Orientated Principles (i.e., Inheritance and Interfaces)

Today’s Practical

- **Programming Exercises (Book):**
  - **Chapter 11.1-11.5 (Only code not UML)**

- Upload single .zip file containing all your java files (only java files).
  - www.zjnu.xyz
  - zip file name should be your student number, e.g., 29392929.zip

- Remember to comment your code, name/student number at the top of files.

- Organise your files so it’s clear to identify each exercise (e.g., file names/folders)

This Week

- Read Associated Chapters
- Review Slides
- Java Exercises
  - Submit Exercise Online
- Online Quizzes
  - Additional quizzes added each week

Questions/Discussion

- Demonstrate your ability to use the IDE/Java in the practical
  - Eclipse
  - Practical/Submission
  - Attendance Sheet