Introduction to Information Systems and Programming

Inheritance and Polymorphism

[some materials adopted from Liang, Introduction to Java Programming]

Objectives

- Inheritance
- Polymorphism & Dynamic Binding
- Casting object
- Protected modifier

Inheritance: motivations

- Think of **circles, rectangles, triangles** classes
- They have common features / properties / behaviors
- What is the best software design to avoid redundancy?

- **Inheritance**
  - Programming constructs to allow **inheriting** code from one class (**superclass**) to another class (**subclass**)
  - Enables you to define a general class (superclass) and later **extend** it to more specialized classes (subclass)

Superclasses and Subclasses

- Example: GeometricObject, Circle
- Class Circle extended from Class GeometricObject
- Use keyword extends
- Superclass / parent class / base class
- Subclass / child class / derived class / extended class
- Subclass inherits all accessible data fields and methods from the superclass, except constructors

Subclass

- In a subclass inherited from a superclass, you can
  - Add new properties
  - Add new methods
  - **Override** the methods of the superclass

Superclasses and Subclasses

- Inheritance is used to model **is-a** relationship (Circle is a GeometricObject)

- Java allows only **single** inheritance: A Java subclass inherits only from one superclass (multiple inheritance can be achieved through interface)

Advantage of inheritance

- Avoid redundancy
  - Different classes may have common properties and behaviors
- Easy to maintain
- Easy to comprehend
  - Class relationship documented in the inheritance tree

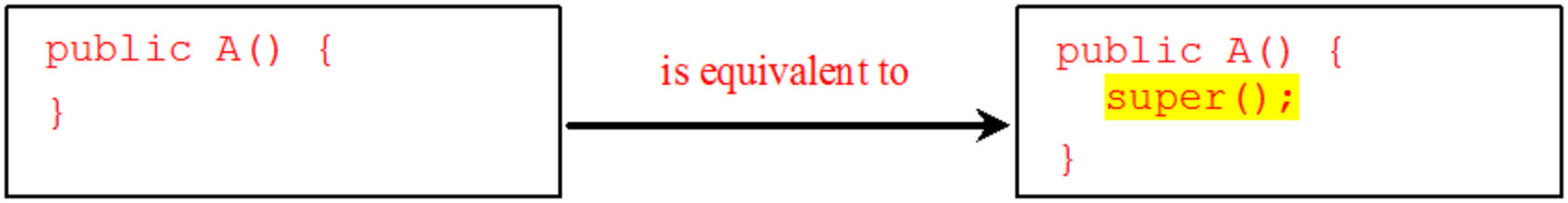
super

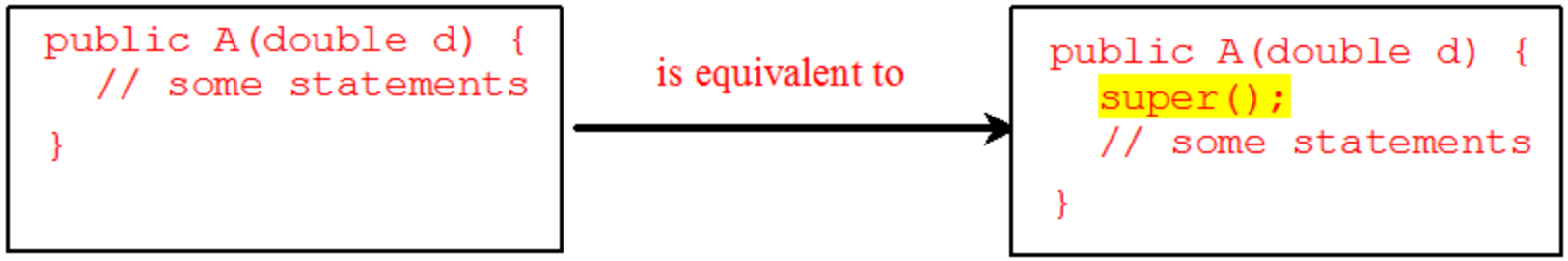
- Superclass's **constructors** are **not** inherited
- However, it can be invoked using super keyword
- The super refers to its superclass object
- If the keyword super is not explicitly used, the superclass's no-arg constructor is implicitly invoked.

Discussion:
Instantiating an object invokes all the superclasses' constructors along the inheritance chain.
True of False?

Superclass's constructor is always invoked

- A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler puts super() as the first statement in the constructor





Java Syntax: *super* & *this*

- To call a superclass constructor
  - Must use super to call the superclass constructor
  - Invoke the superclass constructor's name causes a syntax error
  - super needs to appear first in the constructor
  - Call to constructors (*this()* / *super()*) must be the first statement in the constructor
- Keyword super can also be used to call a superclass method (why do you need this? Aren't they inherited?)

Overriding Methods

- Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass

```
public class Circle extends GeometricObject {
  // Other methods are omitted


  /** Override the toString method defined in GeometricObject */
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

Overriding Methods

- An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside of its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

- You can only override instance methods. You can hide instance attributes / private methods/ static methods / static attributes (overriding vs. hiding)

Code Demo For Inheritance

```java
public class SuperSubclassDemo {
    public static void main(String[] args) {
        Circle c1 = new Circle();
        Circle c2 = new Circle(5.5);
        Circle c3 = new Circle(9.0, "Red");

        System.out.println(c1.getInfo());
        System.out.println(c2.getInfo());
        System.out.println(c3.getInfo());

    }
}

class GeometricObject {
    private String color;
    GeometricObject() {
        this("Green");
    }

    GeometricObject(String color) {
        this.color = color;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
    public String getInfo(){
        return "Geometric Object of color " + this.color;
    }
}
```

```java
class Circle extends GeometricObject {
    private double radius;

    Circle() {
        this(1.0);
    }
    Circle(double radius) {
        this.radius = radius;
    }
    Circle(double radius, String color){
        super(color);
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }
    public void setRadius(double radius) {
        this.radius = radius;
    }
    public double getArea() {
        return Math.PI * Math.pow(this.radius, 2);
    }

    @Override
    public String getInfo() {
        return "Circle with radius " + this.radius + " and color
" + this.getColor();
    }
}
```

Discussion

- Where super() is implicitly invoked?
- Which attributes/methods are inherited and not?
- Do you need @Override to compile the code?
- Why do we need @Override?
- Can you trace the constructor chaining when c1, c2, and c3 are instantiated?

java.lang.Object

- Every class is descended from java.lang.Object
- If no inheritance is specified, the superclass of the class is Object
- Inherited methods from Object, e.g., toString()

```
public class Circle {
   ...
}
```

Equivalent

```
public class Circle extends Object {
   ...
}
```

Polymorphism

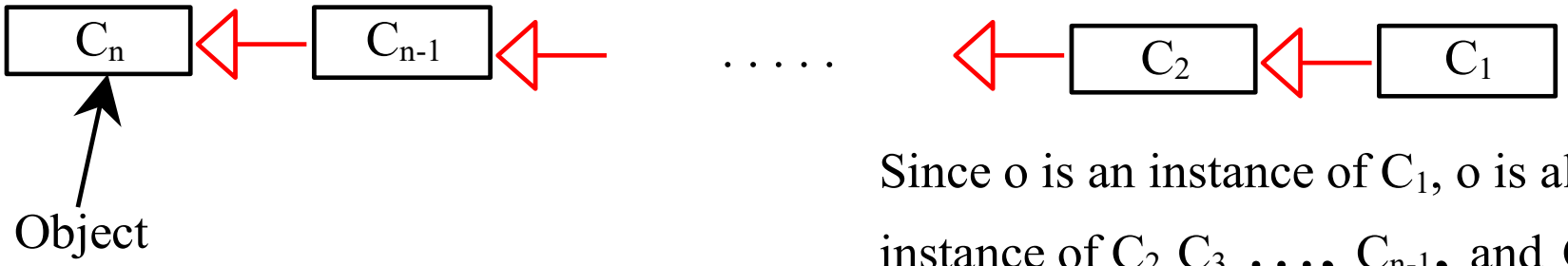
- ”Poly” = many, ”Morph” = form
- Is the implication of inheritance → Multiple classes are related → Single line of code perform different actions

There are 2 types of polymorphism in Java:

1. Compile-time (static): through **inheritance** or **method overloading.** No operator overloading in Java
2. Runtime (dynamic binding): through **method overriding.**

Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n. That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the Object class. If o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked. (*see next slide for the demo)*



E.g., Circle is an Object

Polymorphism, Dynamic Binding

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

Discussion:

1. What happen if the **m** method signature is changed to **m(Person x)**?
2. Where is static polymorphism demonstrated?
3. Where is dynamic binding demonstrated?

Polymorphism, Dynamic Binding

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

Method m takes a parameter of the Object type. You can invoke it with any object.

An **object of a subtype** can be referenced by a **variable of its supertype**. This feature is known as *polymorphism (static)*.

When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method. Which **implementation** is used will be **determined dynamically** by the Java Virtual Machine at runtime. This capability is known as *dynamic binding*.

Every instance of a subclass is also an instance of its superclass, but not vice versa (e.g., Person is an Object, but Object is not a Person)

Casting Object

Casting can be used to convert an object of one class type to another within an inheritance hierarchy. In the preceding section, the statement

```
m(new Student());
```

assigns the object new Student() to a parameter of the Object type. This statement is equivalent to:

```
Object o = new Student(); // Implicit casting
m(o);
```

Down-casting

- Casting from superclass to subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Object y = new Circle();
Circle x = (Circle)y; // Downcasting


Circle z = (Circle) new Object(); //Result in error
```

(Declared type vs actual type)

The instanceof Operator

Use the `instanceof` operator to test whether an object is an instance of a **class**. Can be used to check whether an object implements a certain **interface**:

- Return true if an object is an instance of the type or an instance of a subclass of the type
- Return true if an object implement the interface
- Instanceof cannot be used if the object and the class type are not related in the inheritance line

```
public class InstanceOfDemo {
    public static void main(String[] args) {
        GeometricObject g = new GeometricObject();
        Circle c1 = new Circle();
        String s = "Hello";

        System.out.println( c1 instanceof Circle ); // c1 is a Circle
        System.out.println( c1 instanceof GeometricObject ); // c1 is a subclass of GeometricObject
        System.out.println( c1 instanceof Object ); // c1 is a subclass of Object
        System.out.println( g instanceof Circle ); // g is not a class or subclass of Circle
        System.out.println( s instanceof Object); // s is a sublcass of Object
        System.out.println( s instanceof Circle ); // Error, s and Circle are not related
    }
}
```

Exercise

Downcasting and Instanceof

Consider the following code:

```
class GeometricObject {
}

class Square extends GeometricObject {
    private double side = 5.0;

    public double getSide() {
        return side;
    }
}

class Circle extends GeometricObject {
    private double radius = 2.5;
    public double getRadius() {
        return radius;
    }
}
```

class **Circle** and **Square** are the subclasses of **GeometricObject**

Exercise

Downcasting and Instanceof

Complete the method below to get information of a given GeometricObject object

```
public static String getInfo(GeometricObject g) {
    String s = "";

    // if it is a Circle object, return "Circle with radius of " + circle's radius
    // if it is a Square object, return "Square with a side length of " + square's side length

    return s;
}
```

Test your answer using the code below:

```
public class InstanceOfDemo {
    public static void main(String[] args) {
        Circle c = new Circle();
        Square s = new Square();
        System.out.println( getInfo(c) );
        System.out.println( getInfo(s) );
    }
}
```

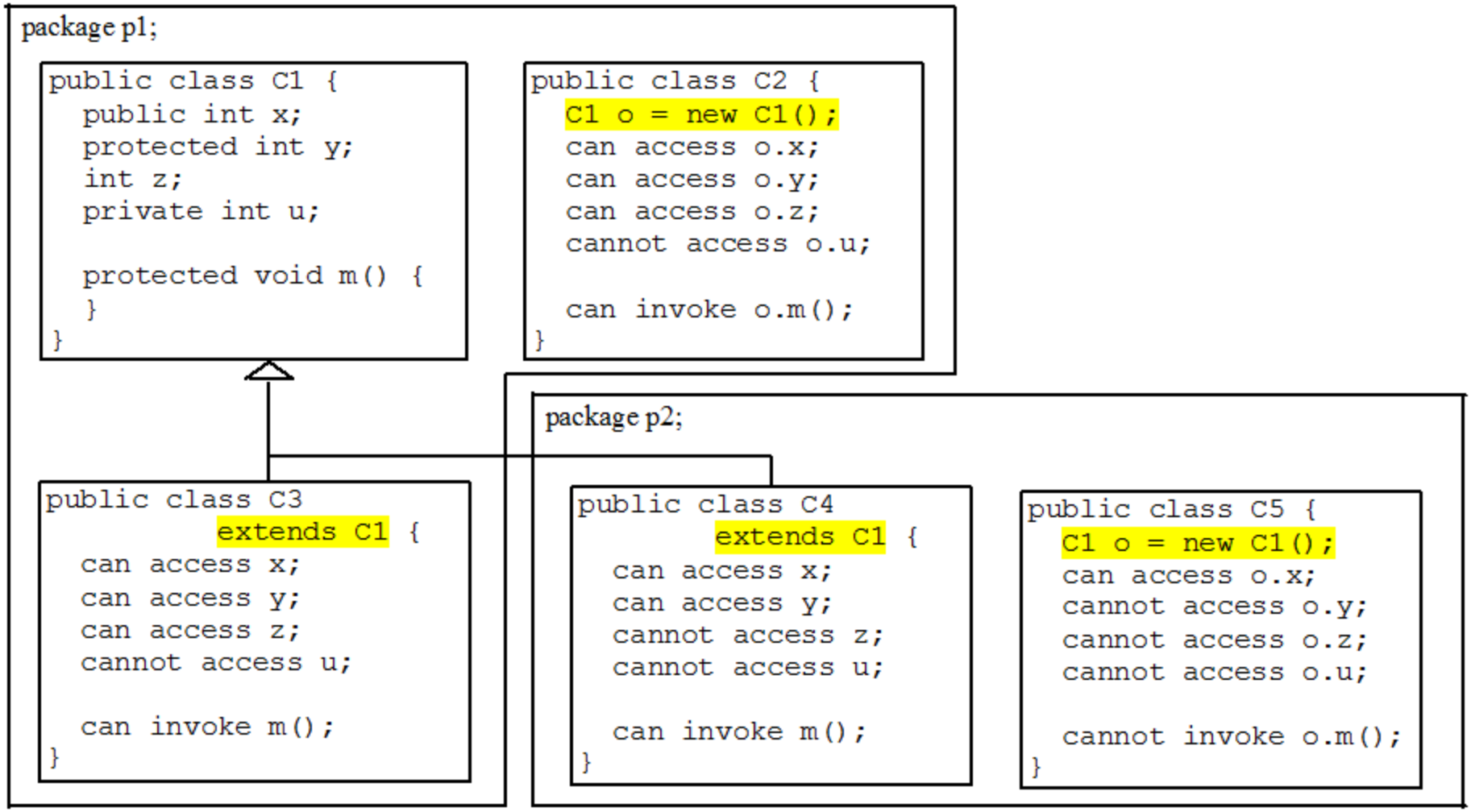
The protected Modifier

- The protected modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package
- 4 access modifiers: private, default, protected, public

Visibility Modifiers

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Visibility Modifiers



A Subclass Cannot Weaken the Accessibility

- A subclass may override a protected method in its superclass and change its visibility to public. However, a subclass cannot weaken the accessibility of a method defined in the superclass. For example, if a method is defined as public in the superclass, it must be defined as public in the subclass