Inheritance

"Question: What is the object oriented way of getting rich?
Answer: Inheritance.“

“Inheritance is new code that reuses old code. Polymorphism is old code that reuses new code.”

Outline

- What is inheritance?
- Inheritance in Java
- Examples: Shape Classes
- The super Keyword
- Access Modifiers

Inheritance

Main Tenets of OO Programming

- Encapsulation
  - abstraction, information hiding
- Inheritance
  - code reuse, specialization "New code using old code."
- Polymorphism
  - do X for a collection of various types of objects, where X is *different* depending on the type of object
  - "Old code using new code."

Things and Relationships

- Object-oriented programming leads to programs that are models
  - sometimes models of things in the real world
  - sometimes models of contrived or imaginary things
- There are many types of relationships between the things in the models
  - chess piece has a position
  - chess piece has a color
  - chess piece moves (changes position)
  - chess piece is taken
  - a rook is a type of chess piece

The “has-a” Relationship

- Objects are often made up of many parts or contain other data.
  - chess piece: position, color
  - die: result, number of sides
- This “has-a” relationship is modeled by composition
  - the instance variables or fields internal to objects
- Encapsulation captures this concept

The “is-a” relationship

- Another type of relationship found in the real world
  - a rook is a chess piece
  - a queen is a chess piece
  - a student is a person
  - a faculty member is a person
  - an undergraduate student is a student
- “is-a” usually denotes some form of specialization
- it is not the same as “has-a”

Inheritance

- The “is-a” relationship is modeled in object oriented languages via *inheritance*
- Classes can inherit from other classes
  - base inheritance in a program on the real world things being modeled
  - does “an A is a B” make sense? Is it logical?

Nomenclature of Inheritance

- In Java the `extends` keyword is used in the class header to specify which preexisting class a new class is inheriting from

```
public class Student extends Person
```

- Person is said to be
  - the parent class of Student
  - the super class of Student
  - the base class of Student
  - an ancestor of Student
- Student is said to be
  - a child class of Person
  - a sub class of Person
  - a derived class of Person
  - a descendant of Person

Results of Inheritance

```
public class A
public class B extends A
```

- the sub class inherits (gains) all instance variables and instance methods of the super class, **automatically**
- additional methods can be added to class B (specialization)
- the sub class can replace (redefine, override) methods from the super class

Question 1

What is the primary reason for using inheritance when programming?

A. To make a program more complicated

B. To duplicate code between classes

C. To reuse pre-existing code

D. To hide implementation details of a class

E. To ensure pre conditions of methods are met.

Question 1

What is the primary reason for using inheritance when programming?

A. To make a program more complicated

B. To duplicate code between classes

C. To reuse pre-existing code

D. To hide implementation details of a class

E. To ensure pre conditions of methods are met.

Inheritance in Java

- Java is a pure object-oriented language
- All code is part of some class
- All classes, except one, must inherit from exactly one other class
- The `Object` class is the *cosmic super class*
  - does not inherit from any other class
  - has several important methods: `toString`, `equals`, `hashCode`, `clone`, `getClass`
- implications:
  - all classes are descendants of `Object`
  - all classes and thus all objects have a `toString`, `equals`, `hashCode`, `clone`, and `getClass` method
    - `toString`, `equals`, `hashCode`, `clone` normally overridden

Inheritance in Java

- If a class header does not include the extends clause, the class extends the `Object` class by default

```
public class Die
```

  - `Object` is an ancestor to all classes
  - it is the only class that does not extend some other class

- A class extends exactly one other class
  - extending two or more classes is *multiple inheritance.* Java does not support this directly, rather it uses *Interfaces*.

Overriding methods

- Any method that is not `final` may be overridden by a descendant class
- Same signature as method in ancestor
- May not reduce visibility
- May use the original method if simply want to add more behavior to existing

Question 2

What is output when the `main` method is run?

```
public class Foo
{
    public static void main(String[] args)
    {
        Foo f1 = new Foo();
        System.out.println( f1.toString() );
    }
}
```

A. `0`

B. `null`

C. Unknown until code is actually run.

D. No output due to a syntax error.

E. No output due to a runtime error.

Shape Classes

- Declare a class called `ClosedShape`
  - assume all shapes have x and y coordinates
  - override `Object`'s version of `toString`
- Possible sub classes of `ClosedShape`
  - `Rectangle`
  - `Circle`
  - `Triangle`
  - `Square`
- Possible hierarchy

  `ClosedShape` **<-** `Rectangle` **<-** `Square`

Defining a Class

- State
  - Class variables
  - Properties
    - Setters and Getters
- Behavior
  - Methods
    - What can it do?
    - What can we tell it to do?
- Identity
  - How distinguish it from other classes?

Defining a Closed Shape Class

- State
  - `int x, y`
  - `getX & setX, getY & setY`
  - `area`
- Behavior
  - `toString`
- Identity
  - constructors

A ClosedShape class

```
public abstract class ClosedShape{
    private double x;
    private double y;

    public ClosedShape()
    {  this(50, 50);     }

    public ClosedShape (double x, double y)
    {  setX(x);
       setY(y);
    }

    public String toString()
    { return "x: " + getX() + " y: " + getY();    }

    public double getX(){ return x; }
    public double getY(){ return y; }

    public abstract double area();

}
```

Constructors with Inheritance

- When creating an object with one or more ancestors, there's a chain of constructor calls
- Reserved word `super` may be used to call a one of the parent's constructors
  - must be first line of constructor
- If no parent constructor is explicitly called the default, calls the default constructor of the parent
  - if no default constructor exists, a syntax error results
- If a parent constructor is called, another constructor in the same class may not be called
  - One or the other, not both
  - good place for an initialization method

Defining a Rectangle Class

- State
  - `int width, height`
  - `getWidth & setWidth, getHeight & setHeight`
  - `area`
- Behavior
  - `toString`
- Identity
  - constructors

A Rectangle Class

```
public class Rectangle extends ClosedShape{

    private double width;
    private double height;

    public Rectangle(){
         this(0, 0);
    }

    public Rectangle(double width, double height){
         setWidth(width);
         setHeight(height);
    }

    public Rectangle(double x, double y,
                 double width, double height){
          super(x, y);
          setWidth(width);
          setHeight(height);
    }

    public String toString(){
         return super.toString() + " width: " + getWidth()
           + " height: " + getHeight();
    }
}
```

The Keyword `super`

- `super` is used to access any protected/public field or method from the super class that has been overridden
- Rectangle's `toString` makes use of the `toString` in ClosedShape my calling `super.toString()`
- Without the super calling `toString` would result in infinite recursive calls
- Java does not allow nested supers

  ```
  super.super.toString()
  ```

  results in a syntax error even though technically this refers to a valid method, `Object`'s toString
- Rectangle *partially* overrides ClosedShapes `toString`

Initialization method

```
public class Rectangle extends ClosedShape{
    private double width;
    private double height;

    public Rectangle(){
        init(0, 0);
    }

    public Rectangle(double width, double height){
        init(width, height);
    }

    public Rectangle(double x, double y,
                double width, double height){
        super(x, y);
        init(width, height);
    }

    private void init(double width, double height){
        setWidth(width);
        setHeight(height);
    }
}
```

Result of Inheritance

Do any of these cause a syntax error? What is the output?

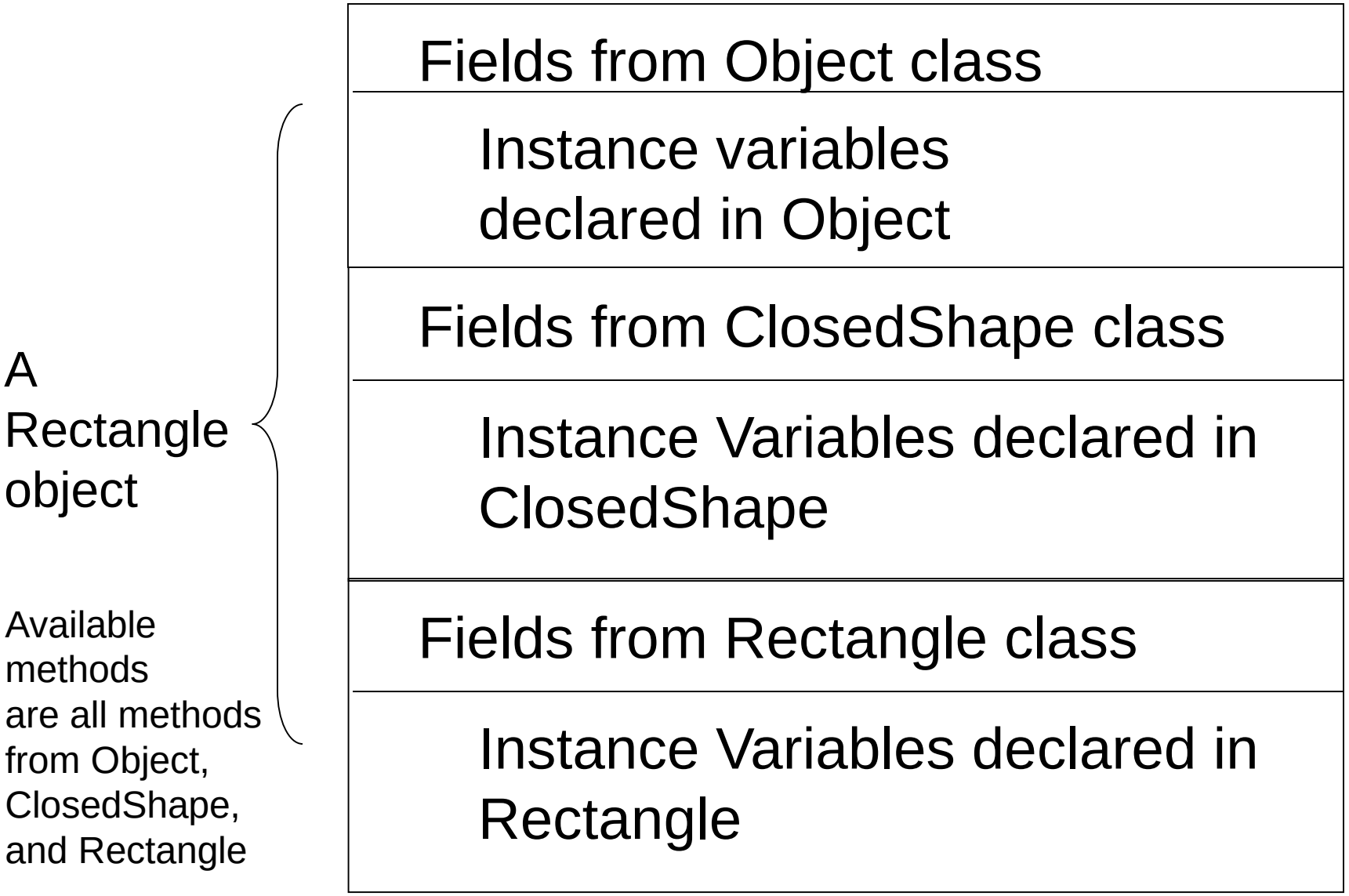
```
Rectangle r = new Rectangle(1, 2, 3, 4);

ClosedShape s = new ClosedShape(2, 3);

s = r;

System.out.println( s.getX() );

System.out.println( s.getWidth() );

System.out.println( s.toString() );

System.out.println( r.getX() );

System.out.println( r.getWidth() );

System.out.println( r.toString() );
```

Result of Inheritance

Do any of these cause a syntax error? What is the output?

```
Rectangle r = new Rectangle(1, 2, 3, 4);

ClosedShape s = new ClosedShape(2, 3);

s = r;

System.out.println( s.getX() );

System.out.println( s.getWidth() );

System.out.println( s.toString() );

System.out.println( r.getX() );

System.out.println( r.getWidth() );

System.out.println( r.toString() );
```

The Real Picture



Access Modifiers and Inheritance

- public
  - accessible to all classes
- private
  - accessible only within that class, hidden from all sub-classes.
- protected
  - accessible by classes within the same *package* and all descendant classes
- Instance variables *should* be private
- protected methods are used to allow descendant classes to modify instance variables in ways other classes can't

Why private vars and not protected?

- In general, it is good practice to make instance variables private
  - hide them from your descendants
  - if you think descendants will need to access them or modify them, provide protected methods to do this
- Why?
- Consider the following example

Required update

```
public class GamePiece {
        private Board board;
        private Position pos;

        // whenever my position changes I must
        // update the board so it knows about the change

        protected void alterPos( Position newPos ){
                Position oldPos = pos;
                pos = newPos;
                myBoard.update( oldPos, pos );
        }
}
```

Why Bother?

- Inheritance allows programs to model relationships in the real world
  - if the program follows the model it may be easier to write
- Inheritance allows code reuse
  - complete programs faster (especially large programs)