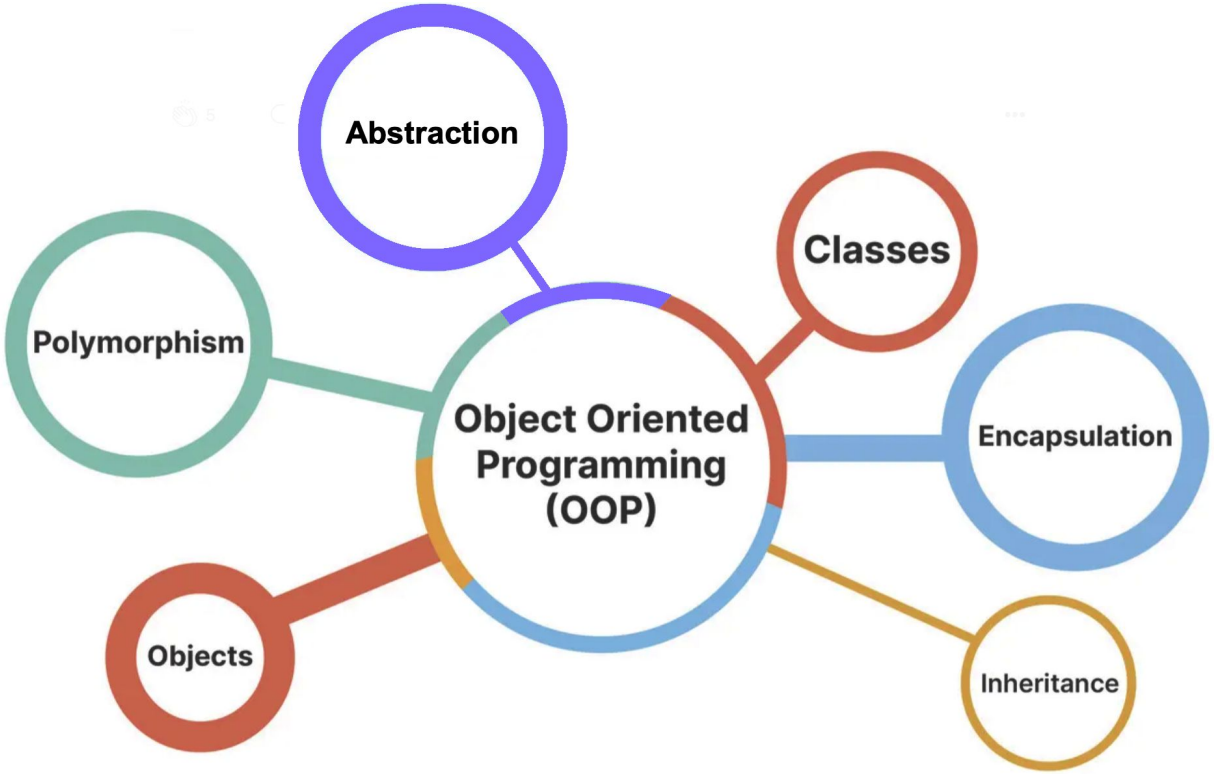
COSC 121: Computer Programming II



Today's Key Concepts

- Polymorphism is a new OOP technique for reference different object types that are related via inheritance or interface
- Implementation techniques:
  - Dynamic binding (as opposed to static binding)
  - The Three Rules
- Relationship to overriding
- Relationship to generic programming
- Using the instanceof operator
- Object casting (just like type casting)

Recall Use of Plus Sign (+)

- Consider the following +:
  - Ex: 4 + 5
    3.14 + 2.9
    str1 + “bar”
    System.out.println("the value is: " + n);
- What does + mean in each case?

Recall Use of Plus Sign (+)

- Consider the following +:
  - Ex: 4 + 5
    3.14 + 2.9
    str1 + “bar”
    System.out.println("the value is: " + n);
- What does + mean in each case?
- How to achieve this?

```
public int +( int x, int y ) {...}
public String +( String x, String y ) {...}
```

What does this remind you of?

Multiple Meanings Across Classes

- What if related classes have methods of the same signature?
- Suppose there are multiple Animal classes, each have their own talk() method implementation:

```
public String talk() { return ""; }     // Animal
public String talk() { return "woof"; }// Dog
public String talk() { return "meow"; }// Cat
public String talk() { return "beh"; } // Sheep
public String talk() { return "moo"; } // Cow
...
```

- Depending on caller object, different talk() would be called

Motivating Example

- Using previous talk() implementation

```
Animal[] myPets = new Animal[4];
                                        // an array of Animal objects
myPets[0] = new Dog();                  // Dog, Cat, Sheep, Cow are
myPets[1] = new Cat();                  // children classes of Animal
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}                                       // new flexibility
```

Motivating Example

- Using previous talk() implementation

```
Animal[] myPets = new Animal[4];

myPets[0] = new Dog();
myPets[1] = new Cat();
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}
```

Output

```
Woof
Meow
Beh
Moo
```

Motivating Example

- Using previous talk() implementation

```
Animal[] myPets = new Animal[4];

myPets[0] = new Dog();
myPets[1] = new Cat();
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}
```

Output

```
Woof
Meow
Beh
Moo
```

at i=0: calls talk() from the **Dog** class

Motivating Example

- Using previous talk() implementation

```
Animal[] myPets = new Animal[4];

myPets[0] = new Dog();
myPets[1] = new Cat();
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}
```

Output

```
Woof
Meow
Beh
Moo
```

at i=1: calls talk() from the **Cat** class

Motivating Example

- Using previous talk() implementation

```
Animal[] myPets = new Animal[4];

myPets[0] = new Dog();
myPets[1] = new Cat();
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}
```

Output

```
Woof
Meow
Beh
Moo
```

at i=1: calls talk() from the **Sheep** class

Motivating Example

- Using previous talk() implementation

```
Animal[] myPets = new Animal[4];

myPets[0] = new Dog();
myPets[1] = new Cat();
myPets[2] = new Sheep();
myPets[3] = new Cow();
for( int i=0; i<myPets.length; i++ )
{
  System.out.println( myPets[i].talk() );
}
```

Output

```
Woof
Meow
Beh
Moo
```

at i=1: calls talk() from the **Cow** class

Polymorphism

- Polymorphism is an OOP technique that allows us to reference different object types at different points in time
  - Literal meaning “having many forms”
- To make use of polymorphism, classes must be related via inheritance or interface relationships (interfaces next topic)
  - In example, Dog, Cat, Sheep, Cow are must all either extends or implements Animal
  - Then reference var of Animal can refer to any of its child objects

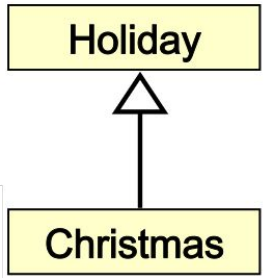
Dynamic Binding

- The call to myPets[i].talk() executes a different method definition in each loop iteration
- Achieved via dynamic binding
  - The call is bound to the definition of the method that it invokes
  - Java defers method binding until run time – so it delays binding until code execution (as late as possible)
  - Note: If this binding occurred at compile time, then that line of code would call the same method every time

General Idea

- An object reference can refer to an object of a related class by inheritance
- Ex: if Holiday is the superclass of Christmas then a Holiday reference could be used to refer to a Christmas object

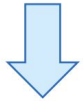
```
Holiday day;
day = new Christmas();
```



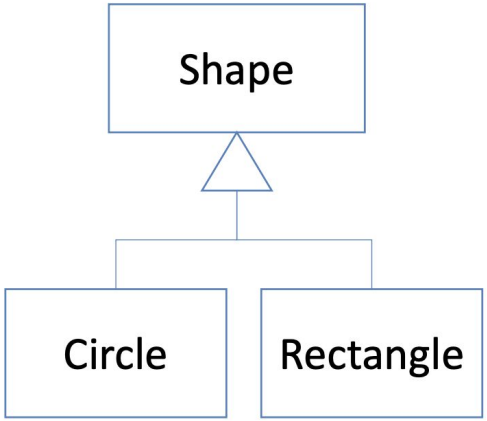
- Can assign Christmas object to a Holiday reference because Christmas is-a Holiday
- To assign child object to parent reference: just use =
- To assign parent object to child reference: need object casting
  (not recommended in practice – afterall, not all holidays are Christmases)

Polymorphism via Inheritance

```
Shape s = new Shape();
System.out.println(s);

Circle c = new Circle(1.5);
System.out.println(c);

Rectangle r = new Rectangle(3.1,2.1);
System.out.println(r);
```



```
Shape s = new Shape();
System.out.println(s);

s = new Circle(1.5);
System.out.println(c);

s = new Rectangle(3.1,2.1);
System.out.println(r);
```



The Three Rules

- Terminology: A class defines a type.
  The type of a subclass (child) is called a subtype.
  The type of a superclass (parent) is called a supertype.
- **Rule 1:** A reference of a supertype can be used to refer to an object of a subtype (but not vice versa).
- **Rule 2:** You can only access class members known to the reference variable.
- **Rule 3:** When invoking a method using a reference variable x, the method in the object referenced by x is executed, regardless of the type of x (this is dynamic binding).

Example Illustrating Rule 1

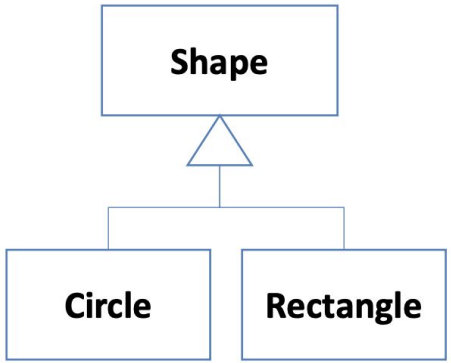
- **Rule 1:** A reference of a supertype can be used to refer to an object of a subtype (but not vice versa). **"Supertype references can point to subtypes"**

Example Illustrating Rule 1

- **Rule 1:** A reference of a supertype can be used to refer to an object of a subtype (but not vice versa). **"Supertype references can point to subtypes"**

All the following statements are valid:

```
Rectange r1 = new Rectangle();
Circle c1 = new Circle();

Shape s1 = new Rectangle();
Shape s2 = r1;
Shape s3 = c1;
```

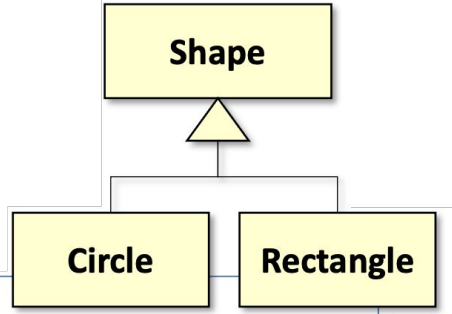


The following statements are **INVALID** :

```
Rectangle r2 = new Shape();  //invalid
Rectangle r3 = new Circle(); //invalid
```

Example 2 Illustrating Rule 1

- We can pass an instance of a subclass to a parameter of its superclass type



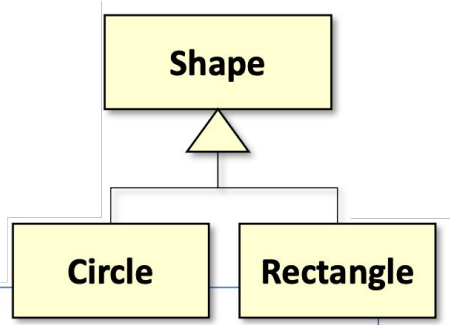
"Supertype references can point to subtypes"

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        Shape s = new Shape("Black", true);
        Circle c = new Circle(10, "Blue", true);
        Rectangle r = new Rectangle(3, 4, "White", false);




    }
    public static void printStatus(Shape sh){
        System.out.println(sh.toString());
    }
}
```

This is OK

Example 2 Illustrating Rule 1

- We can pass an instance of a subclass to a parameter of its superclass type



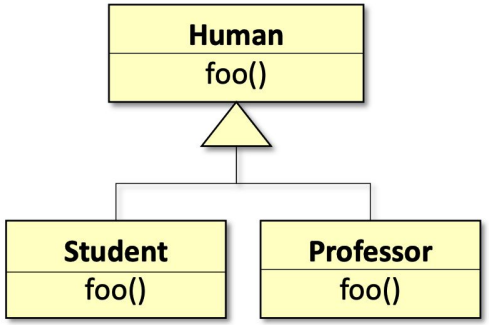
"Supertype references can point to subtypes"

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        Shape s = new Shape("Black", true);
        Circle c = new Circle(10, "Blue", true);
        Rectangle r = new Rectangle(3, 4, "White", false);
        //print the properties of all three shapes
        printStatus(s);
        printStatus(c);
        printStatus(r);
    }
    public static void printStatus(Shape sh){
        System.out.println(sh.toString());
    }
}
```

iClicker Question

What type must go into the purple box in order for the code to be valid?

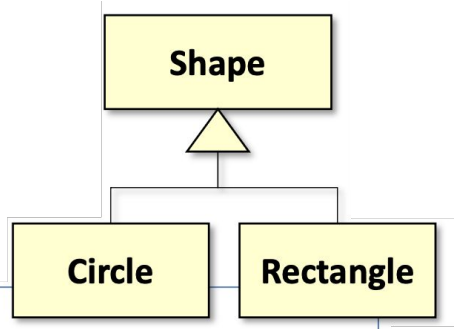
```
          x;
x = new Student();
x.foo();
x = new Professor();
x.foo();
```



A. Human
B. Student
C. Professor
D. None of the above

Example Illustrating Rule 2

- **Rule 2:** You can only access class members **known** to the reference variable.



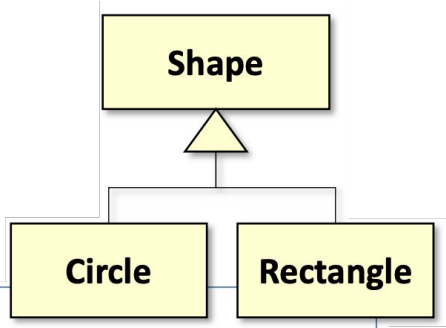

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        Shape s = new Shape("Black", true);
        Circle c = new Circle(10, "Blue", true);
        Rectangle r = new Rectangle(3, 4, "White", false);



    }
    public static void printArea(Shape sh){
        System.out.println(sh.getArea());
    }
}
```

ERROR IF Shape doesn't have getArea()

RULE #2

Example Illustrating Rule 2

- **Rule 2:** You can only access class members **known** to the reference variable.




```
public class PolymorphismDemo {
    public static void main(String[] args) {
        Shape s = new Shape("Black", true);
        Circle c = new Circle(10, "Blue", true);
        Rectangle r = new Rectangle(3, 4, "White", false);

        printArea(c);
        printArea(r);
    }
    public static void printArea(Shape sh){
        System.out.println(sh.getArea());
    }
}
```

c

r

This is OK

ERROR IF Shape doesn't have getArea()

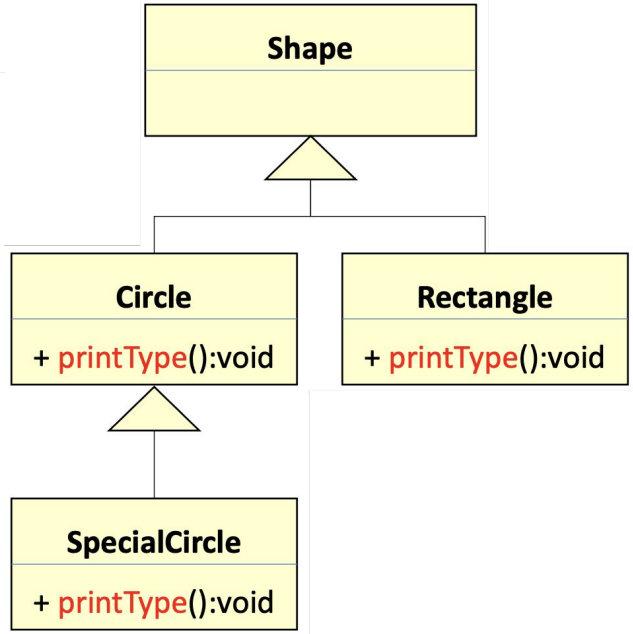
RULE #2

iClicker Question

Assuming the printType() method prints the name of the class, what is the output of the following code?

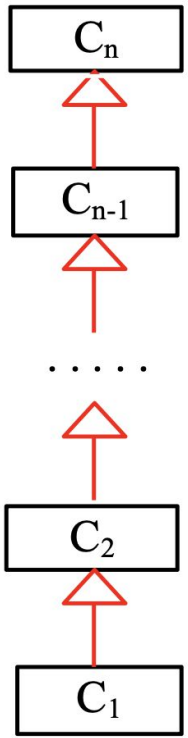
```
Shape x = new Rectangle();
x.printType();
x = new Circle();
x.printType();
x = new SpecialCircle();
x.printType();
```

A. Shape, Shape, Shape
B. Rectangle, Circle, Circle
C. Rectangle, Circle, SpecialCircle
D. Prints something else
E. Error



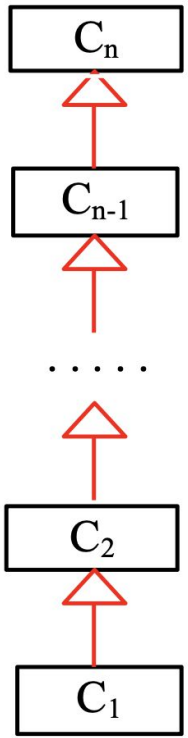
Explanation on Rule 3

- **Rule 3:** When invoking a method using a reference variable x, the method in the object referenced by x is executed, regardless of the type of x (this is dynamic binding).



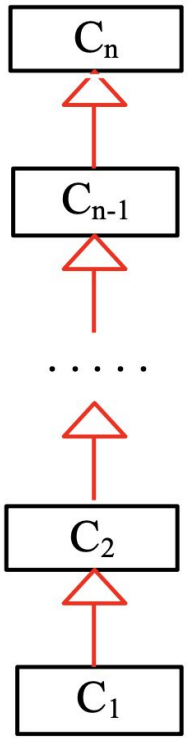
Explanation on Rule 3

- **Rule 3:** When invoking a method using a reference variable x, the method in the object referenced by x is executed, regardless of the type of x (this is dynamic binding).
- Recall class hierarchy (with Object class at top)
- Suppose object obj is an instance of C_1 (and hence it is also an instance of C_2, ..., C_n).



Explanation on Rule 3

- **Rule 3:** When invoking a method using a reference variable x, the method in the object referenced by x is executed, regardless of the type of x (this is dynamic binding).
- Recall class hierarchy (with Object class at top)
- Suppose object obj is an instance of C_1 (and hence it is also an instance of C_2, ..., C_n).
- How dynamic binding works:
  If we invoke a method obj.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.



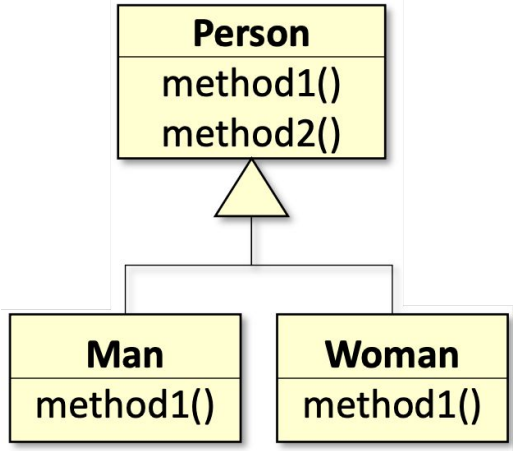
Example Illustrating Rule 3

```
public class Person {
    public void method1(){System.out.println("This is person 1.");}
    public void method2(){System.out.println("This is person 2.");}
}
```

```
public class Man extends Person{
    public void method1(){System.out.println("This is a man.");}
}
```

```
public class Woman extends Person{
    public void method1(){System.out.println("This is a woman.");}
}
```

```
public class DynamicBindingTest {
    public static void main(String[] args) {
        Person p1 = new Person();
        Person p2 = new Man();
        Person p3 = new Woman();
        p1.method1();
        p2.method1();
        p3.method1();
        p1.method2();
        p2.method2();
        p3.method2();
    }
}
```



What output gets printed?

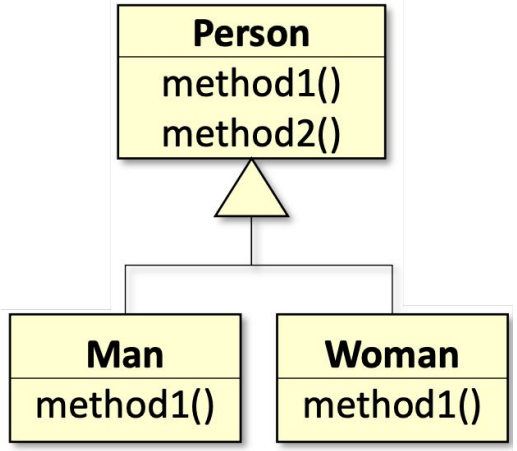
Example Illustrating Rule 3

```
public class Person {
    public void method1(){System.out.println("This is person 1.");}
    public void method2(){System.out.println("This is person 2.");}
}
```

```
public class Man extends Person{
    public void method1(){System.out.println("This is a man.");}
}
```

```
public class Woman extends Person{
    public void method1(){System.out.println("This is a woman.");}
}
```

```
public class DynamicBindingTest {
    public static void main(String[] args) {
        Person p1 = new Person();
        Person p2 = new Man();
        Person p3 = new Woman();
        p1.method1();
        p2.method1();
        p3.method1();
        p1.method2();
        p2.method2();
        p3.method2();
    }
}
```



Output:

```
This is person 1.
This is a man.
This is a woman.
This is person 2.
This is person 2.
This is person 2.
```

Example 2 Illustrating Rule 3

```
public class DynamicBindingTest2 {
    public static void main(String[] args) {
        m(new GradStudent());
        m(new Student());
        m(new Human());
        m(new Object());
    }
    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

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

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

class GradStudent extends Student {
}
```

method m() takes an Object parameter, which means you can invoke it with any object type

- when m() is called, x's toString() is invoked

What output gets printed?

Example 2 Illustrating Rule 3

```
public class DynamicBindingTest2 {
    public static void main(String[] args) {
        m(new GradStudent());
        m(new Student());
        m(new Human());
        m(new Object());
    }
    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

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

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

class GradStudent extends Student {
}
```

method m() takes an Object parameter, which means you can invoke it with any object type

- when m() is called, x's toString() is invoked

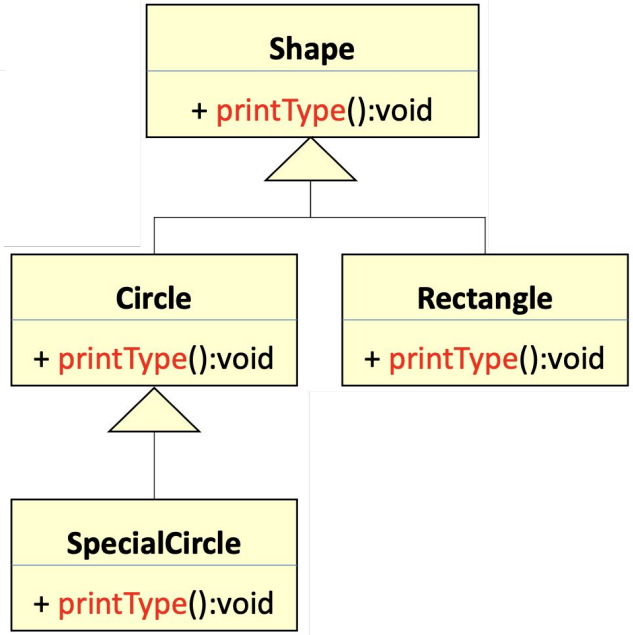
Output:

```
Student
Student
Human
java.lang.Object@5e65ab77
```

iClicker Question

Assuming the printType() method prints the name of the class, what is the output of the following code?

```
Shape x = new Rectangle();
x.printType();
x = new Circle();
x.printType();
x = new SpecialCircle();
x.printType();
```

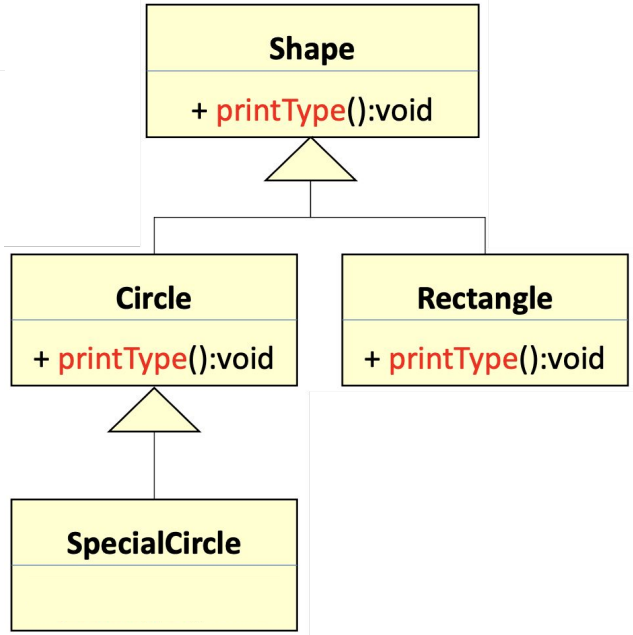


A. Shape, Shape, Shape
B. Rectangle, Circle, Circle
C. Rectangle, Circle, SpecialCircle
D. Prints something else
E. Error

iClicker Question

Assuming the printType() method prints the name of the class, what is the output of the following code?

```
Shape x = new Rectangle();
x.printType();
x = new Circle();
x.printType();
x = new SpecialCircle();
x.printType();
```

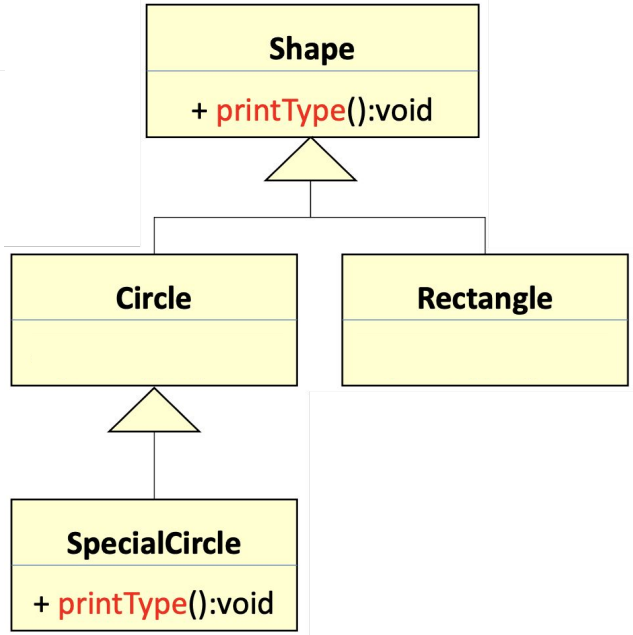


A. Shape, Shape, Shape
B. Rectangle, Circle, Circle
C. Rectangle, Circle, SpecialCircle
D. Prints something else
E. Error

iClicker Question

Assuming the printType() method prints the name of the class, what is the output of the following code?

```
Shape x = new Rectangle();
x.printType();
x = new Circle();
x.printType();
x = new SpecialCircle();
x.printType();
```



A. Shape, Shape, Shape
B. Rectangle, Circle, Circle
C. Rectangle, Circle, SpecialCircle
D. Prints something else
E. Error

Reflecting on Polymorphism

- How does inheritance support polymorphism?

- Is polymorphism same as overriding?

Reflecting on Polymorphism

- How does inheritance support polymorphism?
  - A reference variable of class X can be used to refer to an object of class Y if Y is a descendent of X
  - If both classes contain the same method (i.e., same signature and return type), the parent reference can be polymorphic
- Is polymorphism same as overriding?

Reflecting on Polymorphism

- How does inheritance support polymorphism?
  - A reference variable of class X can be used to refer to an object of class Y if Y is a descendent of X
  - If both classes contain the same method (i.e., same signature and return type), the parent reference can be polymorphic
- Is polymorphism same as overriding?
  - Polymorphism is the ***concept*** that an object has the ability to take many forms
  - Overriding is one of several ***techniques*** to achieve polymorphism

iClicker Question

What is polymorphism in Java?

A. It is when a single parent class has many child classes
B. It is when a program uses several different types of objects, each with its own variable
C. It is when a single variable is used with several different types of related objects at different places in a program
D. It is when a class has several methods with the same name but different parameter types

Relation to Generic Programming

- Recall motivating example:

```
Animal[] myPets = new Animal[4];
// instantiate with various objects
for( int i=0; i<myPets.length; i++ )
  System.out.println( myPets[i].talk() );
```

- Generic programming is a methodology for writing algorithms and data structures using types that are not specified until the algorithm is used
  - Ex: generic sort() function that can order integers, doubles, floats, strings, etc.
- Implementation techniques:
  - Templates (e.g., C++)
  - Generics (e.g., Java Collections Framework)

Next Class:

- Continue with polymorphism
  - Relationship to generic programming
  - Using the instanceof operator
  - Object casting (just like type casting)