

Introduction to Java for C++ Programmers

Segment – 3

JAC 444

Professor: Mahboob Ali

Polymorphism

- Polymorphism enables you to “program in the general” rather than “program in the specific.”
- It’s a concept which leads towards performing a single action by different ways.
- In particular, polymorphism enables you to write programs that process objects that share the same superclass (either directly or indirectly) as if they’re all objects of the superclass; this can simplify programming.

Defining a Contract

- Class defines **contract**



“I have these kind of methods....”

Defining common protocol

- Supertype defines **common protocol (methods)**

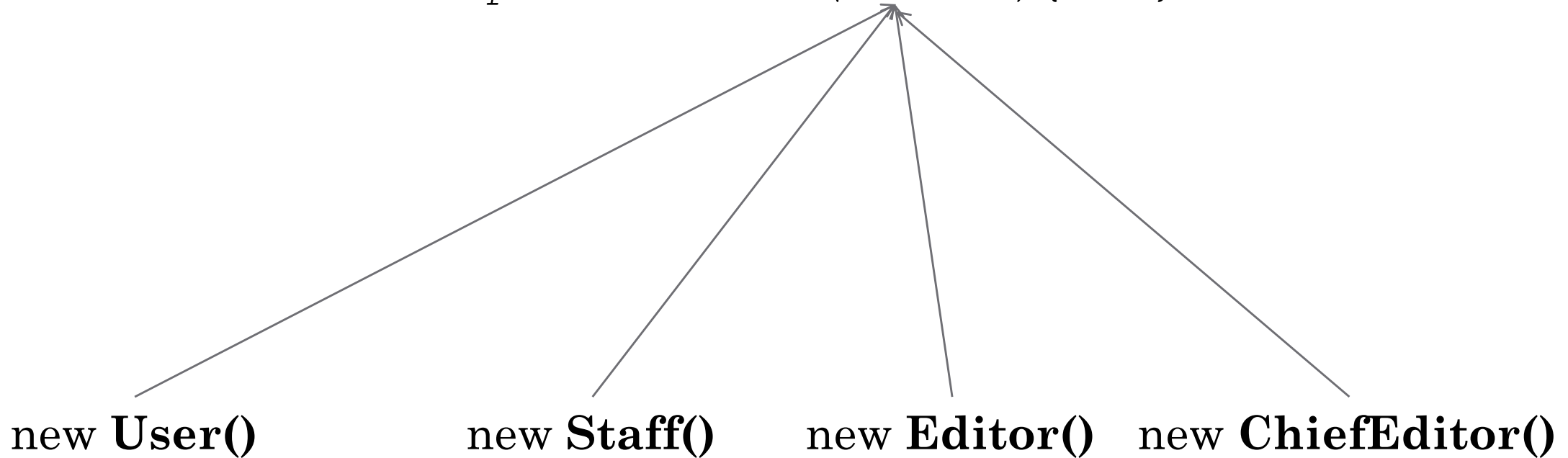


“Myself and my subtypes have the same kind of methods...”

Polymorphism

- **Supertype = subtypes**

```
void updateProfile(User u) { ... }
```



- How can you perform polymorphism?

1. Method overloading

2. Method overriding

```
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 used wherever its supertype value is required. This feature is known as *polymorphism*.

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.

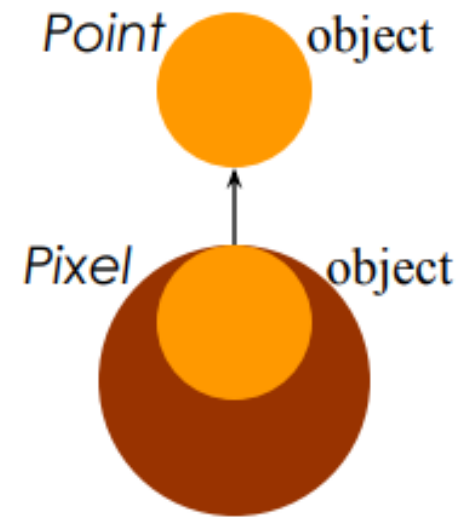
Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime.

This capability is known as *dynamic binding*.

```

class Point {
    int x; int y;
    void clear() { x = 0; y = 0 }
}
class Pixel extends Point{
    Color color;
    public void clear() {
        super.clear();
        color = null;
    }
}

```



Pixel extends both data and behavior of its **Point** superclass.

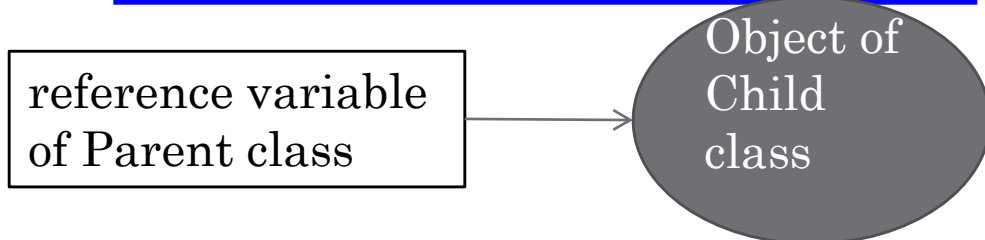
All the **Point** code can be used by anywhere with a **Pixel** in hand.

A single object like **Pixel** could have many (poly) forms (-morph)

It can be used as both a **Pixel** object and a **Point** object.

Pixel's behavior extends **Point**'s behavior.

Point point = new Pixel(); Implicit casting - Upcasting




Casting Objects

Casting can also 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);
```



The statement `Object o = new Student()`, known as implicit casting, is legal because an instance of `Student` is automatically an instance of `Object`.

Why Casting Is Necessary?

Suppose you want to assign the object reference `o` to a variable of the `Student` type using the following statement:

```
Student b = o;
```

A compilation error would occur.

Why does the statement **`Object o = new Student()`** work and the statement **`Student b = o`** doesn't?

This is because a `Student` object is always an instance of `Object`, but an `Object` is not necessarily an instance of `Student`.

```
Student b = (Student)o; // Explicit 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.

```
Apple x = (Apple)fruit;
```

```
Orange x = (Orange)fruit;
```

The instanceof Operator

Use the instanceof operator to test whether an object is an instance of a class:

```
Object myObject = new Circle();  
... // Some lines of code  
/** Perform casting if myObject is an instance of Circle  
 */  
if (myObject instanceof Circle) {  
    System.out.println("The circle diameter is " +  
                        ((Circle)myObject).getDiameter());  
    ...  
}
```

- **Upcasting:** casting to a supertype. Generally you can upcast whenever there is an *is-a* relationship between two classes. Happens automatically in Java, no need to do explicitly.

- a cast from a **Dog** class to an **Animal** class, because a **Dog** *is-a* **Animal**.

- **Downcasting:** casting to a subclass. Java does not do it directly you have to explicitly do it.

- **Animal** animal = new **Dog**();
- **Dog** castedDog = (**Dog**) animal;

Method Matching vs. Binding

- The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time. A method may be implemented in several subclasses.
- The Java Virtual Machine dynamically binds the implementation of the method at runtime.

Example

Polymorphism can be demonstrated with a minor modification to the **Bicycle** class.

For example, a **printDescription** method could be added to the class that displays all the data currently stored in an instance.

```
public void printDescription() {  
    System.out.println("\nBike is " + "in gear " +          this.gear + "  
with a cadence of " + this.cadence          + " and travelling at a speed of "  
+ this.speed      + ". ");  
}
```

To demonstrate polymorphic features in the Java language, extend the **Bicycle** class with a **MountainBike** and a **RoadBike** class.

For **MountainBike**, add a field for **suspension**, which is a **String** value that indicates if the bike has a front shock absorber, **Front**. Or, the bike has a front and back shock absorber, **Dual**.

```
public class MountainBike extends Bicycle {  
    private String suspension;  
    public MountainBike( int startCadence, int startSpeed,  
        int startGear, String suspensionType){  
        super(startCadence, startSpeed, startGear);  
        this.setSuspension(suspensionType);  
    }  
    public String getSuspension(){ return this.suspension; }  
    public void setSuspension(String suspensionType) {  
        this.suspension = suspensionType; }  
    public void printDescription() { super.printDescription();  
        System.out.println("The " + "MountainBike has a" +  
            getSuspension() + " suspension."); }  
}
```

Note the overridden **printDescription** method. In addition to the information provided before, additional data about the suspension is included to the output.

Here is the RoadBike class:

```
public class RoadBike extends Bicycle{
    // In millimeters (mm)
    private int tireWidth;
    public RoadBike(int startCadence, int startSpeed,
        int startGear, int newTireWidth){
        super(startCadence, startSpeed, startGear);
        this.setTireWidth(newTireWidth); }

    public int getTireWidth(){ return this.tireWidth; }

    public void setTireWidth(int newTireWidth){
        this.tireWidth = newTireWidth; }

    public void printDescription(){
        super.printDescription();
        System.out.println("The RoadBike" + " has " +
            getTireWidth() + " MM tires."); } }
```

Note that once again, the **printDescription** method has been overridden. This time, information about the tire width is displayed.

Here is a test program that creates three Bicycle variables. Each variable is assigned to one of the three bicycle classes. Each variable is then printed.

```
public class TestBikes {  
    public static void main(String[] args){  
        Bicycle bike01, bike02, bike03;  
        bike01 = new Bicycle(20, 10, 1);  
        bike02 = new MountainBike(20, 10, 5, "Dual");  
        bike03 = new RoadBike(40, 20, 8, 23);  
        bike01.printDescription();  
        bike02.printDescription();  
        bike03.printDescription(); }  
}
```

The following is the output from the test program:

Bike is in gear 1 with a cadence of 20 and travelling at a speed of 10.

Bike is in gear 5 with a cadence of 20 and travelling at a speed of 10. The MountainBike has a Dual suspension.

Bike is in gear 8 with a cadence of 40 and travelling at a speed of 20. The RoadBike has 23 MM tires.

Keyword: super

- Accessing fields and methods in superclass through object reference:
super

```
class KeySuper {  
    public void methodM() {  
        System.out.println("Coming from Superclass.");  
    }  
}  
  
class Sub extends KeySuper {  
    // overrides methodM in the KeySuper class  
    public void methodM() {  
        super.methodM();  
        System.out.println("Coming from Subclass");  
    }  
}  
  
public class Test{  
    public static void main(String[] args) {  
        Sub x = new KeySuper();  
        x.m(); // what does it print?  
    }  
}
```

```
KeySuper superObj = new Sub();  
if( superObj instanceof Sub)  
    ((Sub) superObj).methodM();
```

Final Classes / Methods

- A class can be declared as final with the declaration:

```
public final class X { ...}
```

- A class that is declared final cannot be subclassed

Example: `java.lang.String`

- A method can be declared as final with the declaration:

```
public class Y {  
    public final void m() {...}  
}
```

A method that is declared final cannot be overridden or hidden by subclasses