Polymorphism

CS110: Introduction to Computer Science

Polymorphism

- **Polymorphism** is a technique that allows us to declare an object as one type but construct it as another.
- This is a challenging concept and that's OK.

What's the problem?

- Let's make an array of every professor and student at AC?
- How do we do this?
- Can we make a SINGLE array containing Students and Professors?
- Right now... no...

Single Array

- Because `Professor` and `Student` both extend `ACPerson` we can make an array of `ACPerson` objects and put both `Professors` and `Students` in it.

```
public static void main(String[] args) {
    ACPerson list[] = new ACPerson[2];
    list[0] = new Student("Nora");
    list[1] = new Professor("Aaron");
}
```

Limitations

- When we create an array of super classes objects, we can only use functions that are in the super class.
- Let's see this in action.

Overridden Methods

- When we access a method that has been overridden, you will access the overridden method rather than the original method.
- Let's see this in action.

Polymorphism

- This technique of declaring a variable or array as a superclass but *defining* it as a subclass is polymorphism.
- Not only does this allow us simplify our data structures but it also lets us create more general functions
- Let's go a make a function that uses polymorphism.
  - We'll make a **`public static`** method for old time sake.

Interfaces

- Sometimes we want the benefits of polymorphism, but we don't want or can't create a super class.
- An **interface** provides a way for us to specify the functions that *must be* included in a class and allows us to use polymorphism.
- It does *not* specify how they implemented.
- The syntax for this is simple.

```java
public interface DrawingObject {
    int getX();
    int getY();
}
```

```java
public class Rectangle implements DrawingObject{

    private int x;
    private int y;
    private int ID;

    public int getX() {
        return x;
    }

    public int getY(){
        return y;
    }

    public Rectangle(int x, int y) {
        this.x = x;
        this.y = y;
        ID = Magic.drawRectangle(x, y, 20, 20, "red");
    }

    public void moveBy(int deltaX, int deltaY) {
        x += deltaX;
        y += deltaY;

        Magic.moveObject(x, y, ID);
    }
}
```

Because of this, if we didn't have **getX()** and **getY()**, we *couldn't compile*

```
public interface DrawingObject {
    int getX();
    int getY();
}
```

```
public class Rectangle implements DrawingObject {

    private int x;
    private int y;
    private int ID;

    public int getX() {
        return x;
    }

    public int getY(){
        return y;
    }

    public Rectangle(int x, int y) {
        this.x = x;
        this.y = y;
        ID = Magic.drawRectangle(x, y, 20, 20, "red");
    }

    public void moveBy(int deltaX, int deltaY) {
        x += deltaX;
        y += deltaY;

        Magic.moveObject(x, y, ID);
    }
}
```

Neat, Why?

- Interfaces allow developers to provide requirements specification.
- We can then use this to create more powerful, flexible code.
- **Let's do an example with Timers**.

Timer

- Java developers have made an object called `Timer` that when constructed and started will execute a function every **`X`** milliseconds.
- But, which function!
- Whatever you want...

Timer Construction

```
public Timer(int delay, ActionListener listener)
```

- `delay` is the number of milliseconds between calls
- `listener` is an object of a class that implements the interface `ActionListener`

`ActionListener`

- The declaration of the `ActionListener` interface is

```
public interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

- So, if we make a class that implements `ActionListener`, then all we need to do is implement `actionPerformed()` with whatever we want to call

Rectangle

- Let's make our rectangle move

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Rectangle implements ActionListener {

    private int x;
    private int y;
    private int ID;

    public Rectangle(int x, int y) {
        this.x = x;
        this.y = y;
        ID = Magic.drawRectangle(x, y, 20, 20, "red");
    }

    public void moveBy(int deltaX, int deltaY) {
        x += deltaX;
        y += deltaY;
       Magic.moveObject(x, y, ID);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        moveBy(1,1);
    }
}
```

Rectangle

- Let's make our rectangle move

The `@Override` is optional, but it speeds up the program.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Rectangle implements ActionListener {

    private int x;
    private int y;
    private int ID;

    public Rectangle(int x, int y) {
        this.x = x;
        this.y = y;
        ID = Magic.drawRectangle(x, y, 20, 20, "red");
    }

    public void moveBy(int deltaX, int deltaY) {
        x += deltaX;
        y += deltaY;
       Magic.moveObject(x, y, ID);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        moveBy(1,1);
    }
}
```

Using Timer

- Now, we can use the start

```
import javax.swing.Timer;

public class Starter {

    public static void main(String[] args){
        Rectangle rect = new Rectangle(20,20);
        Timer t = new Timer(10, rect);
        t.start();
    }
}
```

Using Timer

Java has two Timers.
Make sure to use
`javax.swing.Timer`

- Now, we can use the start

```
import javax.swing.Timer;

public class Starter {

    public static void main(String[] args){
        Rectangle rect = new Rectangle(20,20);
        Timer t = new Timer(10, rect);
        t.start();
    }
}
```

Using Timer

Don't forget to **`start()`**

- Now, we can use the start

```
import javax.swing.Timer;

public class Starter {

    public static void main(String[] args){
        Rectangle rect = new Rectangle(20,20);
        Timer t = new Timer(10, rect);
        t.start();
    }
}
```

Multiple Interfaces

- We can implement multiple interfaces

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Rectangle implements ActionListener, DrawingObject {
    private int x,y,ID;

    public Rectangle(int x, int y) {
        this.x = x;
        this.y = y;
        ID = Magic.drawRectangle(x, y, 20, 20, "red");
    }
    public void moveBy(int deltaX, int deltaY) {
        x += deltaX;
        y += deltaY;
        Magic.moveObject(x, y, ID);
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        moveBy(1,1);
    }
    @Override
    public int getX() {
        return this.x;
    }
    @Override
    public int getY() {
        return this.y;
    }
}
```

Polymorphism in Action

- Consider the following pair of lines

```
public Timer(int delay, ActionListener listener)
```

```
Rectangle rect = new Rectangle(20,20);
Timer t = new Timer(10, rect);
```

- `Timer` requires two parameters: **`int`** and an `ActionListener` object.
- Isn't `rect` a `Rectangle`, can it be an `ActionListener` object too?
- Yes, **this is polymorphism**!

Polymorphism

```
public Timer(int delay, ActionListener listener)
```

```
Rectangle rect = new Rectangle(20,20);
Timer t = new Timer(10, rect);
```

- Because `Rectangle` implements `ActionListener`, any `Rectangle` object can "pretend" to be an `ActionListener` object.
- This act of an object pretending to be another is **polymorphism**.
- To see this in more action, let's make a class `Circle` that implements `ActionListener` and `DrawingObject` and moves by `(-1,-1)` each time

Polymorphism

- As we've seen polymorphism can be used with function parameters, but it can also be used local variables.

```
DrawingObject d1 = new Circle(300,200);
DrawingObject d2 = new Rectangle(100,0);
System.out.println(d1.getX());
System.out.println(d2.getX());
```

- Why would you do this? Mostly for making arrays

```
DrawingObject[] dArr = new DrawingObject[2];
dArr[0] = new Circle(300,200);
dArr[1] = new Rectangle(100,0);
for(DrawingObject element: dArr){
    System.out.println(element.getX());
}
```

Limits of Polymorphism

- Whenever you are utilizing polymorphism, you can **only use the functions declared in the interface you are using**.
- Here is an example of **BAD CODE**.

```
ActionListener a1 = new Circle(300,200);
ActionListener a2 = new Rectangle(100,0);
System.out.println(a1.getX());
System.out.println(a2.getX());
```

- Why is this bad?