Polymorphism

Programming

Learning Objectives

- Previously:
  - Object-oriented programming
  - Inheritance
  - Overriding methods
  - Interfaces
- This week:
  - Polymorphism in Java
    - Classes
    - Interfaces
  - Upcasting and downcasting

Manchester Metropolitan University

Programming

Puzzle Solution

```
static boolean Method1()
{
    System.out.println("Method1 called");
    return false;
}

static boolean Method2()
{
    System.out.println("Method2 called");
    return true;
}

public static void main( String args[] )
{
    if ( Method1() && Method2() )
    {
        System.out.println("Both true");
    }
}
```

Output:

```
Method1 called
```

What happened to Method2? Why was the output not:

```
Method1 called
Method2 called
```

?

Solution: Java stops evaluating a boolean expression once the result is known. In this case, **Method1()** evaluates to **false**, so **Method1() && Method2()** must also be **false**. So, the second method does not need to be called. Similarly a chain of **||**'s will stop at the first **true** value.

Polymorphism

- Polymorphism – from Greek **πολϒσ** (polys - many) **μορφη** (morphe - shapes, forms) – 'many forms'.
- In Biology it is applied to species that can take different forms.
- In object-oriented programming, it refers to accessing objects of different types through a common interface.
- Polymorphism in Java is closely related to the inheritance mechanism.
- We have already used polymorphism – this lecture will consolidate the concept, and introduce some new Java features along the way.

Inheritance - Recap

- Java allows for **single** inheritance via the **extends** keyword.
- The **subclass** or **derived class** inherits data and methods of the **superclass**.
- We can add data and methods to the subclass.
- We can **override** methods in the subclass – this means adding a method with an identical **signature** to one in the superclass.

Example – A Game Engine

- Say we are designing a game engine. We decide to describe all game objects that can be spawned in the world with a class called **GameActor.**
- A **GameActor** has a position (where it is in the world), a method for updating – **void update(float deltaTime)** – and a method for drawing – **void draw()**.
- A **GameActor** also has some number of **components**. These are the things that define the behaviour and appearance of the actor in the world. Examples:
  - **PhysicsComponent** (a component that moves the actor as part of a physics simulation).
  - **MeshComponent** (a mesh of triangles for drawing the object)

(this is a very common way to go about building a game engine).

GameActor class

First cut of the **GameActor** class.

Notes:

- This assumes we have already written a **Vector** class to describe a 3D vector.
- This is simplified – in reality we would have a bit more in here (e.g. rotation, scale).

How do we add the components?

```
class GameActor
{
    Vector pos; // position in the world

    GameActor( Vector initialPos )
    {
        pos = initialPos;
    }

    void update( float deltaTime )
    {
        // tick the actor
    }

    void draw( )
    {
        // render the actor
    }
}
```

Adding Components

Requirements:

1. There are lots of different sorts of components.
2. We might decide to add new component types later in the development – the system needs to be *extensible*.
3. A component can modify the properties of its owning **GameActor** (for instance by moving it).
4. A **GameActor** may have any number of components (including possibly none).
5. Components all have an **update** method, and may have a **draw** method.

Component class

Approach: Define an **ActorComponent** class that encapsulates the behaviour/attributes common to all components, and derive classes from this to describe different types of components.

What would this class look like?

1. What data will it need?
2. What will the constructor look like?
3. What methods will it have?

Component class

- Data – it needs a reference to the owning **GameActor**.
- Constructor – this needs to take a reference to the owning **GameActor** as an argument.
- Methods – **update** method and **draw** method.

```
class ActorComponent
{
    GameActor parent;

    ActorComponent( GameActor parent )
    {
        this.parent = parent;
    }
    void update( float deltaTime )
    {
        // tick the component
    }
    void draw( )
    {
        // do nothing
    }
}
```

Components

Now we can go ahead and write some component classes – here’s two examples.

```
class PhysicsComponent extends ActorComponent
{
    Vector velocity; // a physics component has a velocity
    Vector gravity; // acceleration due to gravity

    PhysicsComponent( GameActor parent )
    {
        super( parent );
        velocity = new Vector( 0.0f, 0.0f, 0.0f );
        gravity = new Vector( 0.0f, 0.0f, -9.8f );
    }
    @Override
    void update( float deltaTime )
    {
        velocity.add( gravity.multiply( deltaTime ) );
        parent.pos.add( velocity.multiply( deltaTime ) );
    }
    // don't override draw - physics component is invisible
}
```

Components

```
class MeshRenderComponent extends ActorComponent
{
    Mesh mesh;

    MeshRenderComponent( GameActor parent, Mesh mesh )
    {
        super( parent );
        this.mesh = mesh;
    }
    @Override
    void update( float deltaTime )
    {
        mesh.updateAnimation( deltaTime );
    }
    @Override
    void draw()
    {
        mesh.renderAtPosition( parent.pos );
    }
}
```

Notes:

- This assumes we have a **Mesh** class – the methods do the obvious things.
- Note the use of **@Override** – this is an optional **annotation** which tells the compiler that we are overriding a method. If the method doesn't exist in the parent class, the compiler will give an error.
- **super** allows us to access methods in the superclass.

Adding to the Actor class

Now to add components to the **GameActor** class. We *could* do it by adding this code to the class.

What are the problems with this approach?

```java
PhysicsComponent physics = null;
MeshRenderComponent meshRender = null;

void AddPhysicsComponent( PhysicsComponent physics )
{
    this.physics = physics;
}
void AddMeshRenderComponent( MeshRenderComponent meshRender )
{
    this.meshRender = meshRender;
}

void update( float deltaTime )
{
    if ( physics != null ) physics.update( deltaTime );
    if ( meshRender != null ) meshRender.update( deltaTime );
}
void draw( )
{
    if ( physics != null ) physics.draw();
    if ( meshRender != null ) meshRender.draw();
}
```

Programming

Problems

1. It’s ugly. We have (potentially) long lists of repeated, very similar code – this is almost always a bad thing that can be solved by better design.
2. It’s not easily extensible. If we think of a new component type, we have to make a component class **and** change the **GameActor** class (for the worse, by making it even uglier).
3. What if we wanted to have two **MeshRender** components to make a composite object? What if we wanted a hundred?

The answer is to use **polymorphism** – each derived component class **is an** actor component so we can refer to it as its native type (e.g. **PhysicsComponent**) *or* as an **ActorComponent.**

Solution

In the **GameActor** class:

```
ArrayList<ActorComponent> componentList;

void AddComponent( ActorComponent component )
{
    componentList.add( component );
}

void update( float deltaTime )
{
    for ( ActorComponent c: componentList )
    {
        c.update(deltaTime);
    }
}

void draw( )
{
    for ( ActorComponent c: componentList )
    {
        c.draw();
    }
}

GameActor( Vector initialPos )
{
    pos = initialPos;
    componentList = new ArrayList<ActorComponent>();
}
```

Much nicer! Now we can add any sort of component, even ones we haven't thought of yet, and they will all be updated and drawn.

We are making use of **polymorphism** – the ability to refer to an object with references of different types.

In Use

```
GameActor myActor = new GameActor( new Vector( 0.0f, 0.0f, 0.0f ));

myActor.AddComponent( new PhysicsComponent( myActor ) );
myActor.AddComponent( new MeshRenderComponent( myActor, new Mesh( "cannonball" )));
```

- Note we are using polymorphism here too – e.g. we pass in a **PhysicsComponent** as a parameter to **AddComponent**. This method expects an **ActorComponent** as a parameter.
- But this is OK, because a **PhysicsComponent** *is an* **ActorComponent**.
- Using **abstraction** (gathering the common behaviour into a superclass) and **polymorphism** we have produced a much clearer, more extensible design.

Polymorphism

- Here, we referred to the same objects using references of different types (an instance of class **PhysicsComponent** was referred to as an **ActorComponent** as well as a **PhysicsComponent.**
- This is **Polymorphism** – an object taking 'different forms'.
- Polymorphism is most powerful when combined with **method overriding** – in this case we call **update** on all of the **ActorComponent** objects, without needing to know what sort of component it is – Java will deal with calling the 'right' version of **update** (this is known as *dynamic dispatch*).

Polymorphism and Interfaces

- Remember that the **implements** keyword (class implements an interface) also represents an 'is-a' relationship.
- So, we can use interface types polymorphically too.

```
interface NoiseMaker
{
    public void MakeNoise();
}

class Dog implements NoiseMaker
{
    public void MakeNoise()
    {
        System.out.println("woof");
    }
}

class Car implements NoiseMaker
{
    public void MakeNoise()
    {
        System.out.println("vroom!");
    }
}
```

```
ArrayList<NoiseMaker>noisyThings
           = new ArrayList<NoiseMaker>();

noisyThings.add( new Dog() );
noisyThings.add( new Car() );

for ( NoiseMaker n : noisyThings )
{
    n.MakeNoise();
}
```

Output

```
woof
vroom!
```

Polymorphism and Interfaces

- In the previous example, we used polymorphism to refer to objects of classes **Dog** and **Car** with references of type **NoiseMaker**.
- This is OK, because a **Dog** *is a* **NoiseMaker** – it implements the **NoiseMaker** interface.
- However, we can only use this reference to a **NoiseMaker** to call methods in the interface – we can't use it to access any of the other data or methods in the object referred to:

```
class Dog implements NoiseMaker
{
    public void MakeNoise()
    {
        System.out.println("woof");
    }
    void Scratch()
    {
        System.out.println("Scratch");
    }
}
```

```
NoiseMaker noisy = new Dog();
noisy.Scratch(); // compile error!
```

Upcasting and Downcasting

- Remember we can use a **cast** to convert between primitive types: e.g.

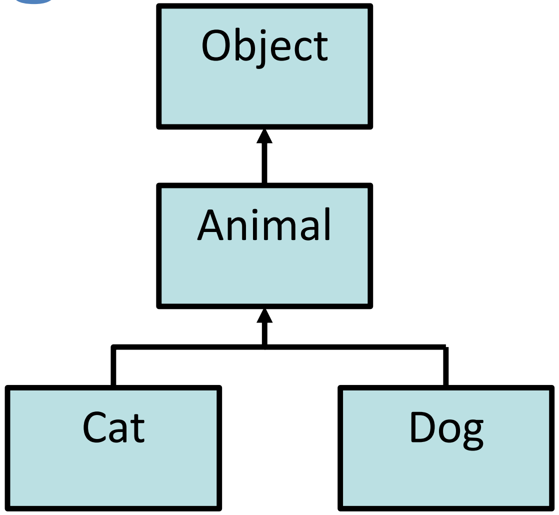
```
int x = 2;
double y = x; // ok - no loss of precision

x = (int)y; // need a cast here - otherwise won't compile.
```

- We can also cast between class types if they are related by inheritance.
- Casting from a class type to a superclass type is called **upcasting** – this happens automatically and doesn't need to be done explicitly.
- Casting from a superclass type to a subclass type is called **downcasting** – this always requires an explicit cast.

Casting Examples

Let’s define a class hierarchy:

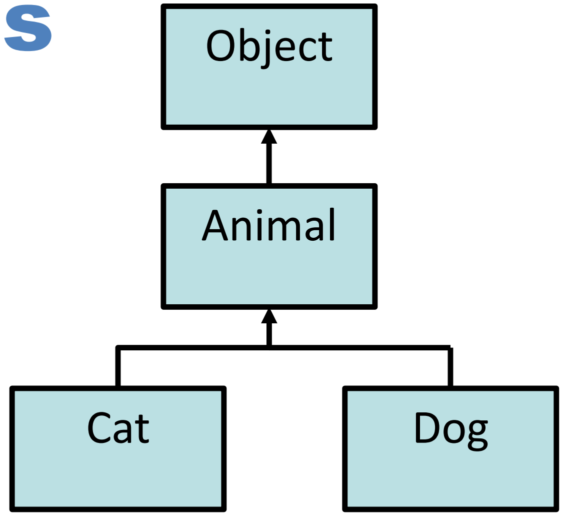
Note that *every* class is derived from **java.lang.Object**



These are all **legal** casts

```
// this is an implicit upcast - a Cat is an Object
Object object = new Cat();

// downcast - this works because the object is
// an instance of Cat (which is an Animal)
Animal animal = (Animal)object;

// another downcast: also works - the object is a Cat
Cat cat = (Cat)animal;
```

Casting Examples



Now for some casts that fail.

```
Cat cat = new Cat();
Dog dog = (Dog)cat;
```

```
Error: java: inconvertible types
  required: Dog
  found:    Cat
```

In this case, the code won't compile.
The compiler knows a **Cat** isn't a **Dog**.

```
Cat cat = new Cat();
Object obj = cat;
Dog dog = (Dog)obj;
```

```
Exception in thread "main"
java.lang.ClassCastException: Cat cannot be cast to
Dog
```

In this case, the code compiles – the reference to **Object** *could* be a **Dog.** Instead, the code fails at runtime and throws a **ClassCastException.**

Quiz

All questions use this hierarchy:

```
interface NoiseMaker
class Animal implements NoiseMaker
class Cat extends Animal
class Dog extends Animal
```

Question 1

```
Cat cat = new Cat();
Dog dog = new Dog();
ArrayList<Object> objectList = new ArrayList<Object>();
objectList.add(cat);
objectList.add(dog);
```

Does this:

(a) Fail to compile
(b) Compile, but throw a runtime exception
(c) Compile and run with no issues

Quiz

All questions use this hierarchy:

```
interface NoiseMaker
class Animal implements NoiseMaker
class Cat extends Animal
class Dog extends Animal
```

Question 1

```
Cat cat = new Cat();
Dog dog = new Dog();
ArrayList<Object> objectList = new ArrayList<Object>();
objectList.add(cat);
objectList.add(dog);
```

Does this:

(a) Fail to compile
(b) Compile, but throw a runtime exception
(c) Compile and run with no issues

This is fine – no cast needed: a **Cat** is an **Object**, as is a **Dog**

Quiz

```
interface NoiseMaker
class Animal implements NoiseMaker
class Cat extends Animal
class Dog extends Animal
```

Question 2

```
Object dogObject = new Dog();
Cat cat = (Cat)dogObject;
```

Does this:

(a) Fail to compile
(b) Compile, but throw a runtime exception
(c) Compile and run with no issues

Programming

Quiz

```
interface NoiseMaker
class Animal implements NoiseMaker
class Cat extends Animal
class Dog extends Animal
```

Question 2

```
Object dogObject = new Dog();
Cat cat = (Cat)dogObject;
```

Does this:

(a) Fail to compile

(b) Compile, but throw a runtime exception

(c) Compile and run with no issues

Compiler can't know that **dogObject** is not a **Cat**, but finds out at runtime.

Quiz

```
interface NoiseMaker
class Animal implements NoiseMaker
class Cat extends Animal
class Dog extends Animal
```

Question 3

```
Object dogObject = new Dog();
ArrayList<Animal> objectList = new ArrayList<Animal>();

objectList.add(dogObject);
```

Does this:

(a) Fail to compile
(b) Compile, but throw a runtime exception
(c) Compile and run with no issues

Quiz

```
interface NoiseMaker
class Animal implements NoiseMaker
class Cat extends Animal
class Dog extends Animal
```

Question 3

```
Object dogObject = new Dog();
ArrayList<Animal> objectList = new ArrayList<Animal>();

objectList.add(dogObject);
```

Does this:

(a) Fail to compile

(b) Compile, but throw a runtime exception

(c) Compile and run with no issues

The compiler doesn't know that **dogObject** is an **Animal** – this needs a downcast to **Animal** (or **Dog**).

Quiz

```
interface NoiseMaker
class Animal implements NoiseMaker
class Cat extends Animal
class Dog extends Animal
```

Question 4

```
Dog dog = new Dog();
Cat cat = new Cat();
ArrayList<NoiseMaker> objectList = new ArrayList<NoiseMaker>();

objectList.add( dog );
objectList.add( cat );
```

Does this:

(a) Fail to compile

(b) Compile, but throw a runtime exception

(c) Compile and run with no issues

Quiz

```
interface NoiseMaker
class Animal implements NoiseMaker
class Cat extends Animal
class Dog extends Animal
```

Question 4

```
Dog dog = new Dog();
Cat cat = new Cat();
ArrayList<NoiseMaker> objectList = new ArrayList<NoiseMaker>();

objectList.add( dog );
objectList.add( cat );
```

Does this:

(a) Fail to compile

(b) Compile, but throw a runtime exception

(c) Compile and run with no issues

This is fine – **Cat** and **Dog** both inherit from **Animal**, which is a **NoiseMaker**, so they are also **NoiseMakers.**

Summary

- Polymorphism, combined with method overriding (dynamic dispatch) is a powerful tool in object-oriented design.
- Interfaces can be used as polymorphic types as well as classes – they can also represent 'is-a' relationships.
- Upcasting (referring to an object with a reference to a superclass) happens automatically.
- Downcasting has to be done explicitly – sometimes the compiler can check this is OK, sometimes not and the code can throw a **ClassCastException**.

Final Puzzle

What's going on here? Read the documentation on **StringBuffer** and see if you can figure it out…

```
StringBuffer text = new StringBuffer('x');
System.out.println("length: "+text.length()+" contents: "+text);
```

Output:

```
length: 0 contents:
```