# 07 Polymorphism

**Three Pillars of OO Programming** – Encapsulation, Inheritance, Polymorphism

## Polymorphic Object Collections

**Inheritance hierarchy design** – tends to be abstract at the top (with a few classes and many abstract methods) and concrete at the bottom (with many classes, and implementations of methods). Consider this hierarchy:

abstract class Animal {

abstract public void speak();

}

abstract class Canine extends Animal {}

abstract class Feline extends Animal {}

class Labrador extends Canine {

public void speak() {

println("Woof!");

}

}

class Chihuahua extends Canine {

public void speak() {

println("Squeak");

}

}

class Lion extends Feline {

public void speak() {

println("Roar!");

}

}

class Balinese extends Feline {

public void speak() {

println("Meow");

}

}

Large number of classes at the bottom of the hierarchy can make management difficult, e.g. suppose we want to keep lists of all animals, then we need four separate lists:

ArrayList<Labrador> labradors

= new ArrayList<Labrador>();

ArrayList<Chihuahua> chihuahuas

= new ArrayList<Chihuahua>();

ArrayList<Lion> lions = new ArrayList<Lion>();

ArrayList<Balinese> balinese = new ArrayList<Balinese>();

Making the objects do something is also difficult:

for (Labrador lab: labradors) lab.speak();

for (Chihuahua ch: chihuahuas) ch.speak();

… etc …

**Polymorphism** – enables objects to *pretend* to be another class. For example, objects lower in an inheritance hierarchy can pretend to be objects higher in the inheritance hierarchy. (This follows because a Labrador is-a Canine which is-a Animal.). This can save write a lot of code, e.g.:

ArrayList<Animal> animals;

void setup() {

// Create the list

animals = new ArrayList<Animal>();

// Populate the list with random animals

/\* …code \*/

// Make each animal speak

for (Animal currentAnimal: animals)

currentAnimal.speak();

}

If a list expects objects of a particular class, it will accept objects of a subclass, e.g.:

animals.add(new Lion());

Balinese myCat = new Balinese();

animals.add(myCat);

**From General Classes to Specialized Classes** – polymorphism only works one way, from the top (superclasses) to the bottom (sublasses) of the hierarchy.

This is evident from this example:

abstract class Animal {

abstract public void speak();

}

class Lion extends Feline {

public void speak() {

println("Roar!");

}

public void attack(){

/\* … attack code …\*/

}

}

All animals can speak but not all animals can attack:

ArrayList<Animal> animals = new ArrayList<Animal>();

animals.add(new Lion());

animals.add(new Chihuahua());

for (Animal animal: animals) {

animal.speak();

animal.attack(); // Error here

}

Polymorphic Object References

Object references are names for an object, not the object itself. Therefore object references may also be polymorphic, e.g.:

Person somePerson;

void setup() {

somePerson = new Employee(); // creates an Employee

somePerson.name = "Elvis";

println(somePerson.name);

}

class Person {

public String name;

}

class Employee extends Person {

public float hourlyRate;

}

The problem now is that properties/methods belonging to the subclass are inaccessible, e.g.

somePerson.hourlyRate = 25.75;

causes an error because Person does not have an hourlyRate property.

**Object casting** – resolves the problem. If you are sure about an object’s “real” type then cast it back when you want subclass methods/properties:

Person somePerson;

void setup() {

somePerson = new Employee();

somePerson.name = "Elvis";

**((Employee)somePerson).hourlyRate = 25.75;**

println(somePerson.name);

**println( ((Employee)somePerson).hourlyRate);**

}

**Warning** – none of this will work if the classes are not related by inheritance!

## Particle System Example

**Particle systems** – common technique in computer graphics used to generate “natural” looking phenomenon such as fire, mist, water etc.