# 08 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!");

}

public void attack() {

println("Attack!");

}

}

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.

The rule is ***-- objects lower in an inheritance hierarchy can “pretend” to be objects higher in the inheritance hierarchy***.

This makes sense because in our common-sense view of the world., we can say that a subclass “is a” superclass. For example:

* a Labrador is a Canine
* a Canine is an Animal
* a Student is a Person
* a Customer is a Person
* a Bike is a Vehicle
* a Car is a Vehicle

Thanks to inheritance, we can be guaranteed that ***subclasses can do everything that their superclass can do***.

For example:

* if a Canine can bark, then so can a Labrador, because Labrador inherits from Canine
* if a Person has an address, then so does a Customer, because Customer inherits from Person
* if a Vehicle has an owner, then so does a Bike, because Bike inherits from Vehicles

Since a subclass can do everything that its superclass can do, then subclasses can “pretend” to be a superclass.

This applies *even if the superclass is abstract*.

Polymorphism 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 \*/

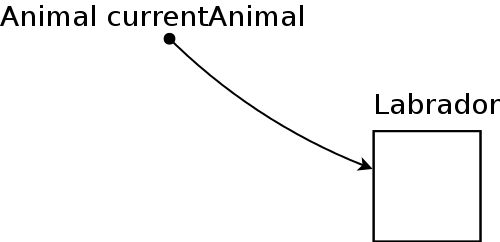
// Make each animal speak

for (Animal currentAnimal: animals)

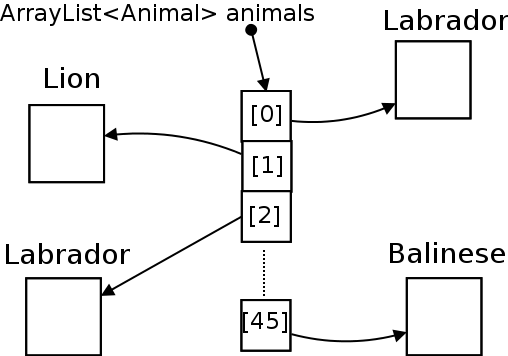
currentAnimal.speak();

}

In the above example, currentAnimal is a reference of type Animal, which is abstract. However the object it refers is from one of Animal’s subclasses, but it is pretending to be an Animal:



Likewise the arraylist is of type ArrayList<Animal> but all the objects it contains are not Animals, but from the subclasses:



Once an inheritance hierarchy is set up, lists of generalized objects can be created and populated from subclasses, 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 …\*/

}

}

The example specifies the following facts:

* all animals can speak
* a Lion is a Feline (which is an Anmial, refer to complete hierarchy)
* a Lion can speak (it must because it inherits speak())
* a Lion can also attack

Although all animals can speak, 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** -- another example:

Person somePerson; // declare a Person

void setup() {

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

somePerson.name = "Elvis";

println(somePerson.name);

}

class Person {

public String name;

}

class Employee extends Person {

public float hourlyRate;

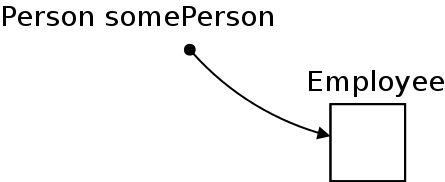
}

One issue for new programmers is that Processing does not know what objects of a subclass can do, and therefore you are not allowed to do those things.

In the following example:

* A Person has a name
* An Employee is a person
* An Employee has an hourly rate

The reference somePerson in the example is of type Person, but it refers to an object of type Employee:



Since somePerson could equally well refer to an object of a different subclass, which does not have an hourlyRate property (e.g. a Manager, who has a salary property instead), this statement causes a compilation error:

somePerson.hourlyRate = 25.75;

Casting

**Object casting** – resolves the problem of reference types and object types not matching. If you know the correct type of an object for sure, then you can “cast” the object to that type in order to specify its properties. Here is the format for a cast, where someObject is of type SuperClass:

SubClass dc = (SubClass)someObject;

For example:

Person somePerson;

void setup() {

somePerson = new Employee();

somePerson.name = "Elvis";

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

println(somePerson.name);

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

}

## Checking an object’s class

Use the instanceof keyword to check the class of an object if you don’t know.

For example, suppose we want to make a function that will only do something with the Lions in a list of Animals:

void makeAllLionsAttack(ArrayList<Animal> manyAnimals){

for (Animal currentAnimal: manyAnimals)

if (**currentAnimal instanceof Lion**)

((Lion)currentAnimal).attack();

}

## Particle System Example

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

**Particle** – an object with an appearance and some properties such as position, velocity etc, which obeys some kind of laws of physics

**ParticleEmitter** – an object that create particles at a specified rate, at a particular position

