
ADVANCED
OBJECT
ORIENTED
PROGRAMMING

ENCAPSULATION, INHERITANCE, POLYMORPHISM, OVERLOADING.

OOP: the basic edition

Up to now, OOP has been awesome for two main reasons:

It lets us organise our code sensible by keeping all those code related to a particular entity or agent or object or concept all together in one place.

It lets us have multiple instances (objects) of the same class which share behaviour and functionality, even while they can behave differently and independently.

Classes

```
class Teacher {
  // Properties (the variables the class has)
  int _iq;

  // Constructor (the setup() of the class)
  Teacher(){
    _iq = 180;
  }

  // Methods (define behaviour/functionality)
  void teach(Student s){
    s._iq = _iq;
  }
}
```

Objects

```
// We use classes like “types”
// which define how an object works
Teacher t;

// We have to make new objects from the class
// with “new” and the constructor
t = new Teacher();
Student s = new Student();

// We call methods defined in the class or
// access properties of the class for this
// specific object with “dot notation”
t.teach(s);
```

OOP: the fancy edition

Object oriented programming involves more than "just" dividing things into classes and using objects in our code.

There are three more concepts central to OO we need to know:

Encapsulation.

Inheritance.

Polymorphism.


ENCAPSULATION

Better keep it encapsulated

A big part of why object-oriented programming even exists is for the purposes of encapsulation.

Classes encapsulate code so that when we use the class (in the form of objects) we don't need to know how they work, we just need to use them.

When we call `t.teach(s)`, we don't need to know how the “teaching” happens, we just need to know that it works.

It also means that the `teach()` method can change internally without us worrying about the details when we call it!

Consider

```
void teach(Student s){
  s._iq = _iq;
}

void teach(Student s){
  s._iq++;
}

void teach(Student s){
 s._iq += int(random(-50,50));
}
```

They will all work

When we use the `teach()` method, all we care about is the idea that something happens to the `Student` object we pass in that means they got taught something.

There are lots of ways that could happen, and we can write the `teach()` method in many different forms.

But because of encapsulation, our main program that uses the method never needs to know about the details.

And so is less likely to break!

Strictness

Encapsulation is actually more strict that we've been treating it in Processing.

One major rule of encapsulation is that you should almost never do anything with an object's properties directly.

So something like `s._iq++`, where we set the `Student` object's `_iq` property directly from the `Teacher` class is bad form.

(OOPs.)

Instead?

Instead of accessing objects' properties directly, we're generally meant to write getters and setters to access them.

These are very simple methods for each class that allow access to properties.

So in the `Student` class we would need something like…

```
class Student {
  int _iq;

  // Omitting other properties and methods

  int getIQ() {
    return _iq;
  }

  void setIQ(int IQ) {
    _iq = IQ;
  }
}
```

```
void teach(Student s){
  s.setIQ(_iq);
}

void teach(Student s){
  s.setIQ(s.getIQ() + 1);
}

void teach(Student s){
 s.setIQ(s.getIQ() + int(random(-50,50)));
}
```

Forcing encapsulation: Privacy

We can force ourselves to encapsulate our objects' properties and methods by using a special word: `private`.

It means that only the class itself is allowed to use those properties and methods.

Any other part of the program that tries to access those properties or call those methods marked private will get an error…

```
class Student {

  private int _iq;

  Student(){
  }

  boolean answerQuestion(Question Q){
    return false;
  }

  private void thinkAboutWeekend(){
    // Ah, the weekend.
  }
}
```

What privacy leads to

```
Student s = new Student();

Teacher t = new Teacher();

Question q = new Question("Do you love math?");

s._iq = 100; // Error! Private!

boolean a = s.answerQuestion(q); // Fine

s.thinkAboutWeekend(); // Error! Private!
```

Publicity

Actually, we can specifically state that properties and methods in our classes are not private by using the special word public.

Using public means that any part of the program can access that property or method directly.

```
public int shirtSize;

public int getIQ(){
   return this.iq;
}
```

Privacy issues?

Generally speaking in Processing, we don't need to worry too much about strict encapsulation.

So if you don't use private properties and methods, and implement getters and setters that's fine.

But it's worth knowing and thinking about the general idea of hiding complexity from yourself that encapsulation is basically all about.

And, in the end, caring about encapsulation will make you a much better programmer.


INHERITANCE

Sometimes objects are quite similar…

A lot of the time when we're writing code, we end up with classes that are quite similar to each other.

So we ended up writing the same code more than once, which feels like a waste of time because...

it is.

Inheritance

The awesome OO programming concept that solves this difficulty is inheritance.

Inheritance allows us to write classes that inherit from or are based on other classes.

Let's write some code about `Cat`s and `Dog`s.

About `Pet`s.

Dogs

```
class Dog {
        int friendly;
        int clean;
        int age;

        Dog(int a) {
            friendly = 10;
            clean = 2;
            age = a;
        }

        void eat() {
            // eating code
        }

        void greeting() {
            println(“WOOF WOOF! WOOF WOOF!”);
        }
}
```

Cats

```
class Cat {
        int friendly;
        int clean;
        int age;

        Cat(int a) {
            friendly = 3;
            clean = 10;
            age = a;
        }

        void eat() {
            // eating code
        }

        void greeting() {
            println(“…”);
        }
}
```

Cats and Dogs are both Pets

Our Cat and Dog code currently has a lot of repetition. What if we wanted to code a Bird or Fish or Slow Loris?

We can make our lives easier via inheritance.

Basically we want to have one class which captures all of the things that will be the same about `Pet`s.

The "super" or "parent" class.

The class that other classes inherit from is called the parent or super class. It will define everything our pets have in common.

Let’s make a generic `Pet` class that does this.

It will be very similar to `Cat` and `Dog` except that it won’t set `friendly` or `clean` values or do anything in `greeting()`

Because that's what we want to change based on the specific pet in question.

Pets

```
class Pet {
      protected int friendly;
      protected int clean;
      protected int age;

      Pet(int a) {
         age = a;
      }

      void eat() {
         // eating code goes here
      }

      void greeting() {
         // do nothing: intentionally leaving this blank
      }
}
```

Protection

Notice that variables of `Pet` are declared `protected`.

The difference between `private` and `protected`:

- a `private` property (or method) can only be used by the class it is in
- a `protected` property (or method) can also be used by any subclasses.

The "child" or "sub" classes.

Now that we have our superclass called `Pet`, we can define two subclasses that inherit from it: `Cat` and `Dog`.

To make inheritance happen, we write that the subclass `extends` the superclass, when writing the class definition.

Because `Cat` and `Dog` will “inherit” the properties and methods of `Pet`, all we need to add to them is a constructor (you just have to) and an appropriate `greeting()` method.

Dogs

```
class Dog extends Pet{

     Dog(int a) {
        super(a);
        super.friendly = 10;
        super.clean = 2;
     }

     void greeting() {
        println(“WOOF WOOF! WOOF WOOF!”);
     }
}
```

Cats

```
class Cat extends Pet {

     Cat(int a) {
        super(a);
        super.friendly = 3;
        super.clean = 10;
     }

     void greeting() {
        println(“…”);
     }
}
```

First things first:
I need to call my parent. super.

Notice that in the constructors for `Cat` and `Dog` we made calls to `super`

That means “do something with my parent’s version of this!” which we need to do to make sure everything gets set up properly.

In a subclass’s constructor, it is implied that the superclass’s constructor will be called. But it is good practice to do it explicitly.

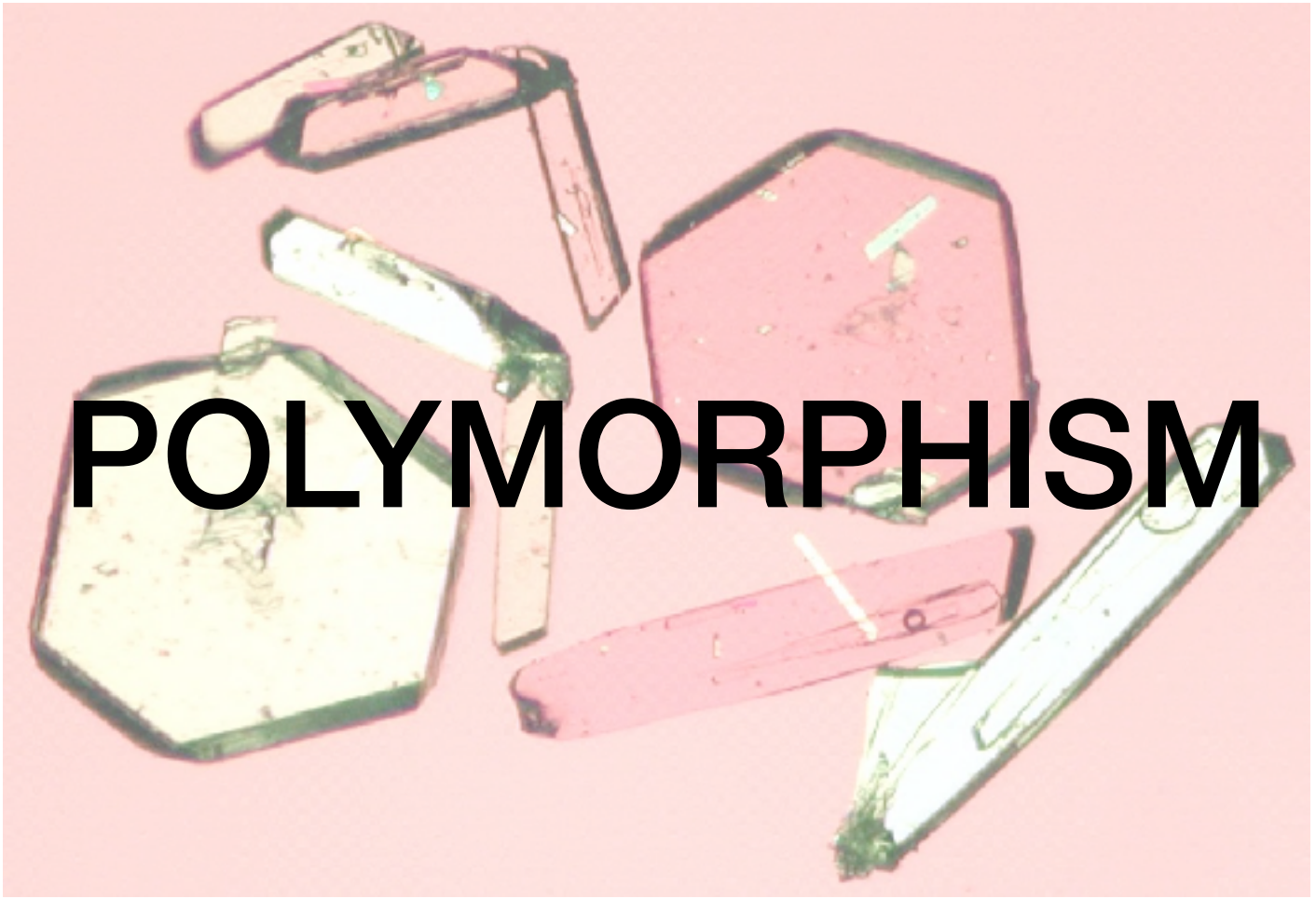
Overriding.

When we wrote our `greeting()` methods in the `Cat` and `Dog` subclasses we were technically "overriding" the same method in the `Pet` class.

You can do this for any method to change how a subclass behaves. When an object of that subclass calls `greeting()` it will call the version in the subclass.

As with constructors, the subclass override version might contain a call to the super class method.

Note: you don't have to override a super class method. For example, `Cat` and `Dog` do not override `Pet`'s eat()


POLYMORPHISM

Suppose we want to make an array of our cats and dogs

Suppose we have a lot of cats and dogs and want to keep track of them.

Before we had the concept of `Pet`, we would have needed to make `Dog`[] arrays and `Cat`[] arrays.

But now that we have the `Pet` class, we can store both our `Cat`s and `Dog`s in an array that holds `Pet`s.

This all works because of polymorphism.

Polymorphism: Big word, simple meaning.

The idea behind it is pretty simple.

The basic point is that when we have something like a `Dog` object, that we know is also a kind of `Pet` object, we can treat it as either of those things.

```
Pet smallFriend = new Dog(3);
```

We can put a `Dog` object anywhere where we would expect a `Pet` variable because Processing knows that `Dog` is a `Pet`.

And it really works!

Pet[] pets = new Pet[20];
pets[0] = new Dog(3);

```
pets[0].greeting();
```

This will use the `greeting()` method from the `Dog` class, even though `pets[0]` — according to our array declaration — looks like it should be of type `Pet`.

This is because Processing knows that `pets[0]` was initialized as a `Dog`, even though we declared it as a `Pet`.

And so…

We can use this feature of polymorphism to make our code even simpler.

Whenever we just want to deal with our subclassed `Pet` objects (e.g. `Cat` or `Dog`) only as generic `Pets`, we can treat them like the `Pet` class,

But with the guarantee that Processing knows what the objects really are: a `Cat` or `Dog`


OVERLOADING

One more thing!

Sometimes we want to have more than one version of the same method.

We might want one constructor that takes no parameters, and another which takes an (x,y) position, for example.

In Processing, we can write both kinds in our programs and use either one, distinguishing them based on which parameters we pass in.

This is called overloading.

```
class Example
{
  private int _x;
  private int _y;

  Example()
  {
    _x = int(random(0, width));
    _y = int(random(0, height));
  }

  Example(int X, int Y)
  {
    _x = X;
    _y = Y;
  }
}
```

And so…

```
Example randomExample = new Example();
```

This code will call the first kind of constructor, which will assign a random position to the object.

```
Example nonrandomExample = new Example(100,100);
```

This code will call the second kind of constructor, which will use the parameters to assign the position.

You've already seen this in fact

```
fill(255);
fill(255, 255, 0);
fill(0, 255, 234, 50);


tint(200);
tint(200, 100);
etc.
```

Works for any kind of method.

You can overload any method (not just constructors).

The only important thing to remember is that each overloaded method has to be different in terms of its parameters: this is known as the method's *signature*. When you call an overloaded method, method signatures are how Processing knows which version you want to use.

Think about the difference between `fill(100)` and `fill(100, 0, 0)`, for instance.