22 Advanced Object-Oriented Programming

"Do you ever think about things that you do think about?"
—Henry Drummond, Inherit the Wind

In this chapter:
– Encapsulation.
– Inheritance.
– Polymorphism.
– Overloading.

In Chapter 8, we introduced *object-oriented programming* ("OOP"). The driving principle of the chapter was the pairing of data and functionality into one single idea, a *class*. A *class* is a template and from that template we made *instances* of *objects* and stored them in variables and arrays. Although we learned how to write classes and make objects, we did not delve very deeply into the core principles of OOP and explore its advanced features. Now that we are nearing the end of the book (and about to leap into the world of Java in the next chapter), it is a good time to reflect on the past and take steps toward the future.

Object-oriented programming in *Processing* and Java is defined by three fundamental concepts: *encapsulation*, *inheritance*, and *polymorphism*. We are familiar with *encapsulation* already; we just have not formalized our understanding of the concept and used the terminology. *Inheritance* and *polymorphism* are completely new concepts we will cover in this chapter. (At the end of this chapter, we will also take a quick look at method *overloading*, which allows objects to have more than one way of calling the constructor.)

22.1 Encapsulation

To understand *encapsulation*, let's return to the example of a *Car* class. And let's take that example out into the world and think about a real-life Car object, operated by a real-life driver: *you*. It is a nice summer day and you are tired of programming and opt to head to the beach for the weekend. Traffic permitting, you are hoping for a nice drive where you will turn the steering wheel a bunch of times, press on the gas and brakes, and fiddle with dial on the radio.

This car that you are driving is *encapsulated*. All you have to do to drive is operate the functions: ***steer()***, ***gas()***, ***brake()***, and ***radio()***. Do you know what is under the hood? How the catalytic converter connects to the engine or how the engine connects to the intercooler? What valves and wires and gears and belts do what? Sure, if you are an experienced auto mechanic you might be able to answer these questions, but the point is you *don't have to* in order to drive the car. This is *encapsulation*.

Encapsulation is defined as hiding the inner workings of an object from the user of that object.

In terms of object-oriented programming, the "inner workings" of an object are the data (variables of that object) and functions. The "user" of the object is you, the programmer, who is making object instances and using them throughout your code.

Now, why is this a good idea? Chapter 8 (and all of the OOP examples throughout the book) emphasized the principles of modularity and reusability. Meaning, if you already figured out how to program a Car, why do it over and over again each time you have to make a Car? Just organize it all into the Car class, and you have saved yourself a lot of headaches.

Encapsulation goes a step further. OOP does not just help you organize your code, it *protects you from making mistakes*. If you do not mess with the wiring of the car while you are driving it, you are less likely to break the car. Of course, sometimes a car breaks down and you need to fix it, but this requires opening the hood and looking at the code inside the class itself.

Take the following example. Let's say you are writing a *BankAccount* class which has a floating point variable for the bank account *balance*.

```
BankAccount account = new BankAccount(1000);
```

A bank account object with a starting balance of $1,000.

You will want to encapsulate that balance and keep it hidden. Why? Let's say you need to withdraw money from that account. So you subtract $100 from that account.

```
account.balance = account.balance - 100.
```

Withdrawing $100 by accessing the balance variable directly.

But what if there is a fee to withdraw money? Say, $1.25? We are going to get fired from our bank programming job pretty quickly, having left this detail out. With encapsulation, we would keep our job, having written the following code instead.

```
account.withdraw(100);
```

Withdrawing $100 by calling a method!

If the ***withdraw()*** function is written correctly, we will never forget the fee, since it will happen every single time the method is called.

```
void withdraw(float amount) {
  float fee = 1.25;
  account -= (amount + fee);
}
```

This function ensures that a fee is also deducted whenever the money is withdrawn.

Another benefit of this strategy is that if the bank decides to raise the fee to $1.50, it can simply adjust the fee variable inside the BankAccount class and everything will keep working!

Technically speaking, in order to follow the principles of encapsulation, variables inside of a class should *never* be accessed directly and can only be retrieved with a method. This is why you will often see programmers use a lot of functions known as *getters* and *setters*, functions that retrieve or change the value of variables. Here is an example of a Point class with two variables (*x* and *y*) that are accessed with getters and setters.

```
class Point {
  float x,y;

  Point(float tempX, float tempY) {
    x = tempX;
    y = tempY;
  }

  // Getters
  float getX() {
    return x;
  }

  float getY() {
    return y;
  }

  // Setters
  float setX(float val) {
    x = val;
  }

  float setY(float val) {
    if (val > height) val = height;
    y = val;
  }
}
```

The variables *x* and *y* are accessed with getter and setter functions.

Getters and setters allow us to protect the value of variables. For example, here the value of *y* can never be set to greater than the sketch's height.

If we wanted to make a point and increment the *y* value, we would have to do it this way:

```
p.setY(p.getY() + 1);
```

Instead of:

```
p.y = p.y + 1;
```

Java actually allows us to mark a variable as "private" to make it illegal to access it directly (in other words, if you try to, the program will not even run).

```
class Point {
  private float x;
  private float y;
}
```

Although rarely seen in *Processing* examples, variables can be set as private, meaning only accessible inside the class itself. By default all variables and functions in *Processing* are "public."

While encapsulation is a core principle of object-oriented programming and is very important when designing large-scale applications programmed by a team of developers, sticking to the letter of the law (as with incrementing the *y* value of the Point object above) is often rather inconvenient and almost silly for a simple *Processing* sketch. So, it is not the end of the world if you make a Point class and access the *x* and *y* variables directly. We have done this several times in the examples found in this book.

However, understanding the principle of *encapsulation* should be a driving force in how you design your objects and manage your code. Any time you start to expose the inner workings of an object outside of the class itself, you should ask yourself: Is this necessary? Could this code go inside a function inside the class? In the end, you will be a happier programmer and keep that job at the bank.

22.2 Inheritance

Inheritance, the second in our list of three fundamental object-oriented programming concepts, allows us to create new classes that are based on existing classes.

Let's take a look at the world of animals: dogs, cats, monkeys, pandas, wombats, and sea nettles. Arbitrarily, let's begin by programming a *Dog* class. A Dog object will have an age variable (an integer), as well as ***eat()***, ***sleep()***, and ***bark()*** functions.

```
class Dog {
  int age;

  Dog() {
    age = 0;
  }

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

  void sleep() {
    // sleeping code goes here
  }

  void bark() {
    println("WOOF!");
  }
}
```

Finished with dogs, we can now move on to cats.

```
class Cat {
  int age;

  Cat() {
    age = 0;
  }

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

  void sleep() {
    // sleeping code goes here
  }

  void meow() {
    println("MEOW!");
  }
}
```

Notice how dogs and cats have the same variables (age) and functions (eat, sleep). However, they also have a unique function for barking or meowing.

Sadly, as we move onto fish, horses, koala bears, and lemurs, this process will become rather tedious as we rewrite the same code over and over again. What if, instead, we could develop a generic *Animal* class to describe any type of animal? After all, all animals eat and sleep. We could then say the following:

- A dog is an animal and has all the properties of animals and can do all the things animals can do. In addition, a dog can bark.
- A cat is an animal and has all the properties of animals and can do all the things animals can do. In addition, a cat can meow.

Inheritance allows us to program just this. With *inheritance*, classes can inherit properties (variables) and functionality (methods) from other classes. The Dog class is a child (AKA a *subclass*) of the Animal class. Children inherit all variables and functions automatically from their parent (AKA *superclass*). Children can also include additional variables and functions not found in the parent. Inheritance follows a tree-structure (much like a phylogenetic "tree of life"). Dogs can inherit from Canines which inherit from Mammals which inherit from Animals, and so on. See Figure 22.1.

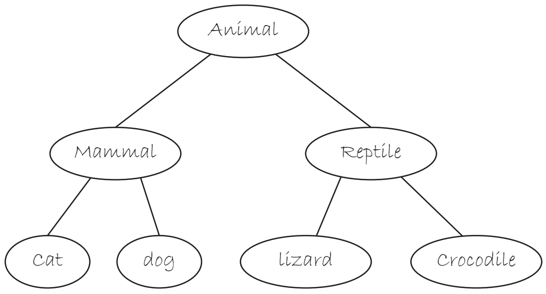

fig. 22.1

Here is how the syntax works with inheritance.

```
class Animal {
  int age;

  Animal() {
    age = 0;
  }

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

  void sleep() {
    // sleeping code goes here
  }
}

class Dog extends Animal{
  Dog() {
    super();
  }
  void bark() {

    println("WOOF!");
  }
}
```

The Animal class is the parent (or super) class.

The variable age and the functions ***eat()*** and ***sleep()*** are inherited by Dog and Cat.

The Dog class is the child (or sub) class. This is indicated with the code "***extends Animal***"

```
class Cat extends Animal{
  Cat() {
    super();
  }
  void bark() {
    println("MEOW!");
  }
}
```

super() means execute code found in the parent class.

Since ***bark()*** is not part of the parent class, we have to define it in the child class.

The following new terms have been introduced:

- ***extends***—This keyword is used to indicate a parent class for the class being defined. Note that classes can only extend *one* class. However, classes can extend classes that extend other classes, that is, Dog extends Animal, Terrier extends Dog. Everything is inherited all the way down the line.
- ***super()***—Super calls the constructor in the parent class. In other words, whatever you do in the parent constructor, do so in the child constructor as well. This is not required, but is fairly common (assuming you want child objects to be created in the same manner as their parents). Other code can be written into the constructor in addition to ***super()***.

A subclass can be expanded to include additional functions and properties beyond what is contained in the superclass. For example, let's assume that a Dog object has a hair color variable in addition to age, which is set randomly in the constructor. The class would now look like so:

```
class Dog extends Animal {
  color haircolor;

  Dog() {
    super();
    haircolor = color(random(255));
  }

  void bark() {
    println("WOOF!");
  }
}
```

A child class can introduce new variables not included in the parent.

Note how the parent constructor is called via ***super()***, setting the age to 0, but the haircolor is set inside the Dog constructor itself. Suppose a Dog object eats differently than a generic Animal. Parent functions can be *overridden* by rewriting the function inside the subclass.

```
class Dog extends Animal {
  color haircolor;

  Dog() {
    super();
    haircolor = color(random(255));
}

  void eat() {
    // Code for how a dog specifically eats
  }

  void bark() {
    println("WOOF!");
  }
}
```

A child can ***override*** a parent function if necessary.

But what if a Dog should eat the same way an Animal does, but with some additional functionality? A subclass can both run the code from a parent class and incorporate some custom code.

```
class Dog extends Animal {
  color haircolor;
  Dog() {
    super();
    haircolor = color(random(255));
  }

  void eat() {
    // Call eat() from Animal
    super.eat();
    // Add some additional code
    // for how a dog specifically eats
    println("Yum!!!");
  }

  void bark() {
    println("WOOF!");
  }
}
```

A child can execute a function from the parent while adding its own code as well.

Exercise 22-1: Continuing with the Car example from our discussion of encapsulation, how would you design a system of classes for vehicles (that is, cars, trucks, buses, and motorcycles)? What variables and functions would you include in a parent class? And what would be added or overridden in the child classes? What if you wanted to include planes, trains, and boats in this example as well? Diagram it, modeled after Figure 22.1.

22.3 An Inheritance Example: SHAPES

Now that we have had an introduction to the theory of inheritance and its syntax, we can develop a working example in *Processing*.

A typical example of inheritance involves shapes. Although a bit of a cliche, it is useful because of its simplicity. We will create a generic "Shape" class where all Shape objects have an *x*, *y* location as well as a size, and a function for display. Shapes move around the screen by "jiggling" randomly.

```
class Shape {
  float x;
  float y;
  float r;

  Shape(float x_, float y_, float r_) {
    x = x_;
    y = y_;
    r = r_;
  }

  void jiggle() {
    x += random(-1,1);
    y += random(-1,1);
  }

  void display() {
    point(x,y);
  }
}
```

A generic shape does not really know how to be displayed. This will be overridden in the child classes.

Next, we create a subclass from Shape (let's call it "Square"). It will inherit all the instance variables and methods from Shape. We write a new constructor with the name "Square" and execute the code from the parent class by calling ***super()***.

Variables are inherited from the parent.

```
class Square extends Shape {

  // we could add variables for only Square here if we so desire

  Square(float x_, float y_, float r_) {
    super(x_,y_,r_);
  }

  // Inherits jiggle() from parent

  // Add a display method
  void display() {
    rectMode(CENTER);
    fill(175);
    stroke(0);
    rect(x,y,r,r);
  }
}
```

If the parent constructor takes arguments then ***super()*** needs to pass in those arguments.

Aha, the square overrides its parent for display.

Notice that if we call the parent constructor with ***super()***, we must have to include the required arguments. Also, because we want to display the square onscreen, we override ***display()***. Even though we want the square to jiggle, we do not need to write the ***jiggle()*** function since it is inherited.

What if we want a subclass of Shape to include additional functionality? Following is an example of a Circle class that, in addition to extending Shape, contains an instance variable to keep track of color. (Note this is purely to demonstrate this feature of inheritance, it would be more logical to place a color variable in the parent Shape class.) It also expands the ***jiggle()*** function to adjust size and incorporates a new function to change color.

```
class Circle extends Shape {

  // inherits all instance variables from parent + adding one
  color c;

  Circle(float x_, float y_, float r_, color c_) {
    super(x_,y_,r_);  // call the parent constructor
    c = c_;           // also deal with this new instance variable
  }

  // call the parent jiggle, but do some more stuff too
  void jiggle() {
    super.jiggle();
    r += random(-1,1);
    r = constrain(r,0,100);
  }

  void changeColor() {
    c = color(random(255));
  }

  void display() {
    ellipseMode(CENTER);
    fill(c);
    stroke(0);
    ellipse(x,y,r,r);
  }
}
```

The Circle jiggles its size as well as its *x,y* location.

The ***changeColor()*** function is unique to circles.

To demonstrate that inheritance is working, here is a program that makes one Square object and one Circle object. The Shape, Square, and Circle classes are not included again, but are identical to the ones above. See Example 22-1.

Example 22-1: Inheritance

```
// Object oriented programming allows us to define classes in terms of other classes.
// A class can be a subclass (aka "child") of a super class (aka "parent").
// This is a simple example demonstrating this concept, known as "inheritance."

Square s;
Circle c;

void setup() {
  size(200,200);
  smooth();

  // A square and circle
  s = new Square(75,75,10);
  c = new Circle(125,125,20,color(175));
}

void draw() {
  background(255);

  c.jiggle();
  s.jiggle();

  c.display();
  s.display();
}
```

This sketch includes one Circle object and one Square object. No Shape object is made. The Shape class just functions as part of our inheritance tree!

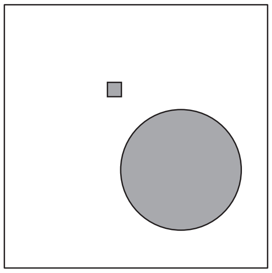
fig. 22.2

Exercise 22-2: Write a Line class that extends Shape and has variables for the two points of the line. When the line jiggles, move both points. You will not need "r" for anything in the line class (but what could you use it for?).

```
class Line _______ {

  float x2,y2;

  Line(_______,_______,_______,_______) {

    super(_________________);

    x2 = _______;

    y2 = _______;
  }

  void jiggle() {

    ______________________

    ______________________

    ______________________

  }

  void display() {
    stroke(255);
    line(______________________);
  }
}
```

Exercise 22-3: Do any of the sketches you have created merit the use of inheritance? Try to find one and revise it.

22.4 Polymorphism

Armed with the concepts of inheritance, we can program a diverse animal kingdom with arrays of dogs, cats, turtles, and kiwis frolicking about.

```
Dog[] dogs = new Dog[100];
Cat[] cats = new Cat[101];
Turtle[] turtles = new Turtle[23];
Kiwi[] kiwis = new Kiwi[6];
```

100 dogs. 101 cats. 23 turtles. 6 kiwis.

```
for (int i = 0; i < dogs.length; i++) {
  dogs[i] = new Dog();
}
for (int i = 0; i < cats.length; i++) {
  cats[i] = new Cat();
}
for (int i = 0; i < turtles.length; i++) {
  turtle[i] = new Turtle();
}
for (int i = 0; i < kiwis.length; i++) {
  kiwis[i] = new Kiwi();
}
```

Because the arrays are different sizes, we need a separate loop for each array.

As the day begins, the animals are all pretty hungry and are looking to eat. So it is off to looping time.

```
for (int i = 0; i < dogs.length; i++) {
  dogs[i].eat();
}
for (int i = 0; i < cats.length; i++) {
  cats[i].eat();
}
for (int i = 0; i < turtles.length; i++) {
  turtles[i].eat();
}
for (int i = 0; i < kiwis.length; i++) {
  kiwis[i].eat();
}
```

This works great, but as our world expands to include many more animal species, we are going to get stuck writing a lot of individual loops. Isn't this unnecessary? After all, the creatures are all animals, and they all like to eat. Why not just have one array of Animal objects and fill it with all different kinds of Animals?

```
Animal[] kingdom = new Animal[1000];

for (int i = 0; i < kingdom.length; i++) {
  if (i < 100) kingdom[i] = new Dog();
  else if (i < 400) kingdom[i] = new Cat();
  else if (i < 900) kingdom[i] = new Turtle();
  else kingdom[i] = new Kiwi();
}
for (int i = 0; i < kingdom.length; i++) {
  kingdom[i].eat();
}
```

The array is of type ***Animal***, but the elements we put in the array are Dogs, Cats, Turtles, and Kiwis.

When it is time for all the Animals to eat, we can just loop through that one big array.

The ability to treat a Dog object as either a member of the Dog class or the Animal class (its parent) is known as *Polymorphism*, the third tenet of object-oriented programming.

Polymorphism (from the Greek, *polymorphos,* meaning many forms) refers to the treatment of a single object instance in multiple forms. A Dog is certainly a Dog, but since Dog *extends* Animal, it can also be considered an Animal. In code, we can refer to it both ways.

```
Dog rover = new Dog();
Animal spot = new Dog();
```

Normally, the type on the left must match the type on the right. With polymorphism, it is OK as long as the type on the right is a child of the type on the left.

Although the second line of code might initially seem to violate syntax rules, both ways of declaring a Dog object are legal. Even though we declare spot as an Animal, we are really making a Dog object and storing it in the spot variable. And we can safely call all of the Animal methods on spot because the rules of inheritance dictate that a Dog can do anything an Animal can.

What if the Dog class, however, overrides the ***eat()*** function in the Animal class? Even if spot is declared as an Animal, Java will determine that its true identity is that of a Dog and run the appropriate version of the ***eat()*** function.

This is particularly useful when we have an array.

Let's rewrite the Shape example from the previous section to include many Circle objects and many Square objects.

The old "non-polymorphism" way with multiple arrays.

```
// Many Squares and many Circles
Square[] s = new Square[10];
Circle[] c = new Circle[20];

void setup() {
  size(200,200);
  smooth();

  // Initialize the arrays
  for (int i = 0; i < s.length; i++) {
    s[i] = new Square(100,100,10);
  }
  for (int i = 0; i < c.length; i++) {
    c[i] = new Circle(100,100,10,color(random(255),100));
  }
}

void draw() {
  background(100);

  // Jiggle and display all squares
  for (int i = 0; i < s.length; i++) {
    s[i].jiggle();
    s[i].display();
  }

  // Jiggle and display all circles
  for (int i = 0; i < c.length; i++) {
    c[i].jiggle();
    c[i].display();
  }
}
```

Polymorphism allows us to simplify the above by just making one array of Shape objects that contains both Circle objects and Square objects. We do not have to worry about which are which, this will all be taken care of for us! (Also, note that the code for the classes has not changed, so we are not including it here.) See Example 22.2.

Example 22-2: Polymorphism

```
// One array of Shapes
Shape[] shapes = new Shape[30];

void setup() {
  size(200,200);
  smooth();
  for (int i = 0; i < shapes.length; i++) {
    int r = int(random(2));
    // randomly put either circles or squares in our array
    if (r == 0) {
      shapes[i] = new Circle(100,100,10,color(random(255),100));
    } else {
      shapes[i] = new Square(100,100,10);
    }
  }
}

void draw() {
  background(100);
  // Jiggle and display all shapes
  for (int i = 0; i < shapes.length; i++) {
    shapes[i].jiggle();
    shapes[i].display();
  }
}
```

The new polymorphism way with one array that has different types of objects that extend Shape.

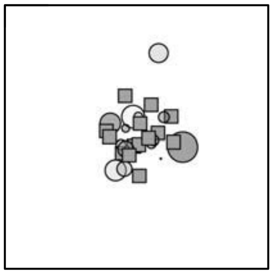
fig. 22.3

Exercise 22-4: Add the Line class you created in Exercise 22-2 to Example 22-2. Randomly put circles, squares, and lines in the array. Notice how you barely have to change any code (you should only have to edit ***setup()*** *above).*

Exercise 22-5: Implement polymorphism in the sketch you made for Exercise 22-3.

22.5 Overloading

In Chapter 16, we learned how to create a Capture object in order to read live images from a video camera. If you looked at the *Processing* reference page (*http://www.processing.org/reference/libraries/video/Capture.html*), you may have noticed that the Capture constructor can be called with three, four, or five arguments:

```
Capture(parent, width, height)
Capture(parent, width, height, fps)
Capture(parent, width, height, name)
Capture(parent, width, height, name, fps)
```

Functions that can take varying numbers of arguments are actually something we saw all the way back in Chapter 1! ***fill()***, for example, can be called with one number (for grayscale), three (for RGB color), or four (to include alpha transparency).

```
fill(255);
fill(255,0,255);
fill(0,0,255,150);
```

The ability to define functions with the same name (but different arguments) is known as *overloading*. With ***fill()***, for example, *Processing* does not get confused about which definition of ***fill()*** to look up, it simply finds the one where the arguments match. A function's name in combination with its arguments is known as the function's ***signature***—it is what makes that function definition unique. Let's look at an example that demonstrates how overloading can be useful.

Let's say you have a ***Fish*** class. Each ***Fish*** object has a location: *x* and *y*.

```
class Fish {
  float x;
  float y;
```

What if, when you make a Fish object, you sometimes want to make one with a random location and sometimes with a specific location. To make this possible, we could write two different constructors:

```
  Fish() {
    x = random(0,width);
    y = random(0,height);
  }

  Fish(float tempX, float tempY) {
    x = tempX;
    y = tempY;
  }
}
```

Overloading allows us to define two constructors for the same object (as long as these constructors take different arguments).

When you create a Fish object in the main program, you can do so with either constructor:

```
Fish fish1 = new Fish();

Fish fish2 = new Fish(100,200);
```

If two constructors are defined, an object can be initialized using either one.