

CS 116x Winter 2015
Craig S. Kaplan

Module 07

Advanced types and Object-Oriented Programming

Topics

- Working with objects
- Defining custom classes
- Collections
- Using standard collection types

Readings

- *Getting Started*, Pages 129-140
- *Learning Processing*, Chapter 8 (see also the [online version](#))



Introduction

For the first time in a while, I'd like to introduce a few new ways to write programs. That is, I won't be talking about new libraries that give us extra functionality (like `ControlP5` or `Fisica`), nor will I be explaining the effective use of built-in functions (like `pushMatrix()` or `popMatrix()`). Instead, we're actually going to grow the syntax of the Processing (well, Java) language itself. There won't suddenly be new programs we can write that we couldn't have written before by other means. But new constructs give us opportunities to *model* the solutions to problems in different, sometimes more elegant, ways. And some programming problems can be solved *much* more elegantly with the help of Object-Oriented Programming. I'll also introduce some new built-in

types to handle collections of data. For the most part, that won't require adding to the language, though we may require a few small tweaks to get around places where Processing doesn't work elegantly with objects.



Understanding Objects

Objects have snuck into the course in a few places so far. At the time I made an effort to gloss over the “objectness” of those ideas, because getting into all the details was a can of worms that would distract from the topic being discussed. Here are some clear signs that you're working with objects:

- You're using types that start with capital letters (such as `String`, `PImage`, `PShape`, `FWorld`, and `Button`). There's no *requirement* that object types be given capitalized type names. It's a convention—we all agree to do it because it makes code easier to read.
- You put a dot between two words. For example, writing `img.width`, `myarray.length`, or `a_circle.setvelocity()`. The “dot” operator allows you to reach into an object and either read a piece of data from it or send a message to it.
- You use the special keyword `new`. I'm not talking specifically about arrays; for example, we used `new` a lot when working with `Fisica`. Arrays are a bit special in Processing, though in fact they're also objects.
- You use the magic word `this`. A tiny piece of my soul dies every time I have to invoke `this` without explanation.

In fact, objects are a lot more pervasive than that. It turns out that everything you do in Processing is running inside an object! Processing just does a good job of hiding this fact from you so that you don't have to think about it.

When you first start learning programming in Processing, it's safe to ignore most of the object-related stuff. But as more and more of it creeps into examples and libraries, it becomes increasingly

important to understand it. There's also a more direct benefit: programming with objects is a powerful and elegant technique that will make you a better programmer.

With that in mind, let's review what we already know about objects based on code we've seen in the course so far.

- At the highest level, objects behave like any other values in Processing. Squint your eyes for a second and think about the absolutely generic things you can do with values that don't depend on the specifics of their types. You can assign them to variables, put them in arrays, pass them to functions, get them back from functions, and so on. These behaviours are obvious with simple types like `int` and `float`. But objects can do the same things. We've already seen examples in the course of arrays of `PImage`, `String` variables, and so on.
- It looks like objects can aggregate data together: a single object can somehow have multiple values inside of it. For example, given a variable of type `PImage`, you can ask for its `width`, its `height`, and its array of `pixels`. Note that every distinct `PImage` can have its unique set of values inside of it. If I have an array of `PImages`, I can get ten different `ints` when I ask each one for its `width`.
- Objects also have *behaviour*. I've spoken so far this term about *sending messages* to objects. Sometimes these messages are a way to tell the object to do something, to change the program but not provide some information back as a result. In Fisica, when we send the `setVelocity()` message to an `FBody`, something changes inside the object behind the scenes. Some messages are more about querying the object for information. When we send a `PImage` the `get()` message, we're asking it to tell us something about itself without modifying itself. A message can take a combination of both of these forms, but it can be helpful to think of messages as being primarily about modifying the object, or primarily about getting information from it.
- We can create objects using the `new` keyword. This was most obvious when using the Fisica library,

since it was our responsibility to create all physical objects and then add them to the world. ControlP5 hides `new` from us, but it's using it at some point to generate objects representing sliders, buttons, checkboxes, and so on.

Why is all of this so useful? There are a few standard reasons why object-oriented programming is considered a Good Idea.

- With objects, we can aggregate together multiple pieces of data that form a cohesive whole. So far in this course, when we've wanted to keep track of, say, the X and Y coordinates of a point, we've been forced to use two separate variables (or two separate, parallel arrays of variables). But conceptually, a 2D point is a single object that should be treated as having its own distinct identity. OO lets us "bind" multiple pieces of data together this way. As a side benefit, it becomes possible to return multiple values from a function, by returning an object containing all the values of interest.
- We can associate behaviour (functions) more tightly with the pieces of data that the functions operate on. This helps create a *separation of concerns*, and clarifies the breakdown of code in large programs.
- The messages supported by an object act as a kind of public interface to that function's data, an approved way to deal with the information it contains. Using an object's public interface avoids having to reach in and fiddle with low-level data directly.
- Objects are just one more way to think about solving problems. Some real-world problems are most easily expressed by mapping the main pieces of the problem into objects.

Of course, there's more. Object-oriented programming is a very large topic, and there are many other advanced aspects to it that we won't talk about in this course.



Object-oriented programming terminology

Let's lay down the specialized language of object-oriented programming up front, and use the terms in the rest of the module.

- **Class:** the type associated with a set of related objects. `String`, `PImage` and `Button` are all classes. “Primitive” types like `int`, `float` and `boolean` aren't classes—these basic types don't have the properties of objects.
- **Instance:** an object belonging to a particular class. In a line of code like

```
PImage img = loadImage( "cat.jpg" );
```

we would say that `img` is an instance of the class `PImage`.

- **Field:** a value that lives inside an instance. The `width` and `height` of an image are fields of the class `PImage`.
- **Method:** a function (“message”) that operates on an instance.
- **Constructor:** a special method that is used to initialize a new instance of a class. The constructor is the method that is invoked when you use `new`.



Writing classes

A class is a way to group together a collection of related values and methods. In fact, it introduces a new *scope* in which those values and methods co-exist. It makes sense, then, that a class declaration

surrounds most of the meat of the class inside of curly braces, like a function:

```
class BestClassEver
{
    // Fields
    // Constructors
    // Methods
}
```

Every class declaration starts with the keyword `class` followed by the name of the class to be defined (usually capitalized, by convention). Then, inside the curly braces, we can introduce fields and methods, including constructors. Let's do that in a second; first, it's good to pause and notice that even a trivial class already behaves like other types in Processing. In the code below we introduce a new class `T` with no fields and no methods. Instances of this class have no real behaviour or state—they don't hold any information and don't compute anything for you. But they do have their identity as members of the type `T`, which means that you can do things like create them, make arrays of them, pass them to functions, and so on.

```
class T {}

void doSomething( T t )
{
    println( "Yep, it's a T." );
}

void setup()
{
    T my_t = new T();
    T[] array_of_t = new T[ 27 ];
    doSomething( my_t );
}
```

Now, a small piece of good news: that's almost the only new syntax we need in order to invent new classes. Field declarations inside a class look just like variable declarations. Method declarations look just like function declarations. There are just a few remaining additions:

- A constructor looks like a method, but it doesn't have a return type, and its name is required to be the same as the name of the surrounding class. It's used to initialize a freshly minted instance. A constructor can take any number of arguments (or none at all), just like any function or method. If you don't include a constructor for a class, Processing will automatically create a zero-argument "do-nothing" constructor for you.

In fact, you can have multiple distinct constructors in the same class, as long as Processing can tell them apart based on the types of the arguments.

```
class Test
{
  int x;

  // Constructor
  Test( int some_x )
  {
    x = some_x;
  }
}
```

- We use `new` to ask Processing to set aside space for a new instance of a class, and to invoke the constructor to initialize that instance.

```
Test my_test = new Test( 13 );
```

- There's a special value called `null`, which is implicitly compatible with every class. It means "undefined", and can be assigned to variables or passed to functions. It's an error to do anything with `null` beyond naming it.

```

void setup()
{
    // Totally fine: assign p to be
    // invalid for now.
    Point p = null;
    // ERROR! Can't manipulate a field
    // of the null value.
    p.x = 3;

    // OK: you're always allowed to ask
    // if a variable holds null or a
    // real
    // value.
    if( p == null ) {
        // The variable used to be
        // invalid, but after this it
        // will point to an actual
        // instance of Point.
        p = new Point( 3, 4 );
    }
    // Now that p is valid, it's OK to
    // do things with it like call a
    // method.
    println( p.mag() );
}

```

- There's also the mysterious keyword `this`, but I'm going to hold off on explaining that for just a bit longer.

The key to understanding classes is that the inside of a class is a new *scope*. Recall that a scope is a region of the program in which a specific set of declarations is visible. When you create an instance of a class using `new`, you get a fresh copy of all the fields, bound together in one object. When you call a method of a class, that method gets to manipulate the data of one particular instance of the class. Which one? The one that you called the method on (i.e., the one you sent the message to, as I said in earlier modules). At the point where you call the method, that's the variable name (or more generally, the expression) to the left of the dot. Similarly, the fields of different instances are entirely distinct from each other.

Let's put all of that together into a simple, complete class.

```
class NamedNumber
{
    float val;
    String name;

    NamedNumber(
        float a_val, String a_name )
    {
        val = a_val;
        name = a_name;
    }

    void report()
    {
        println( name, "=", val );
    }
}

void setup()
{
    NamedNumber n1 =
        new NamedNumber( PI, "PI" );
    NamedNumber n2 =
        new NamedNumber( 2.718, "e" );
    NamedNumber n3 =
        new NamedNumber( 6, "Six" );

    n1.report();
    n2.report();
    println( n1.val - n2.val );
}
```

Objects are especially useful when you need to maintain a lot of copies of similar-looking data. For example, it's far more elegant to create a `Point` class and use an array of `Points` than to deal with two separate, parallel arrays of X and Y coordinates. This insight can be applied to sketches throughout the term in which we dealt with multiple arrays in parallel.

```

void setup()
{
  Point[] pts = new Point[ 10 ];
  for( int idx = 0;
      idx < pts.length; ++idx ) {
    pts[idx] =
      new Point( idx, 100 * sin( idx ) );
  }

  float total = 0;
  for( int idx = 0; idx < 10; ++idx ) {
    total += pts[idx].mag();
  }

  println( total );
}

```

Unfortunately, some of the built-in Processing functions for arrays, like `append()` and `shorten()`, don't work quite right with arrays of instances. See the documentation for those functions, which explains how to work around this deficiency.

Example sketch: DirectManipMultiOO

Example sketch: BlizzardOO

Example sketch: FoamOO

It's not that objects can *only* be used in conjunction with collections like arrays. You can declare a class and create instances of it wherever you want. It's just that if you only plan to use one or two instances of an object, the fields you have in mind might just as easily be made into global variables and methods made into global functions. However, another place where you might implicitly want to track a lot of copies of an object would be when using recursion. For example, a `Point` class could be used in conjunction with recursion to simplify the construction of a fractal like the Sierpinski gasket or the Pinwheel tiling. The result might use roughly the same number of lines of code, but is perhaps a little cleaner and easier to think about.

Example sketch: CornerGasketOO

The body of a method lives implicitly within a single instance of the class, the instance that first received the method call (though of course it may have indirect access to other instances if they're passed in as arguments or visible globally). Every now and then we want to be able to refer to *the instance itself*, i.e., the surrounding object that we think we're inside of. That, at last, is the meaning of `this`. The name is defined only inside class scope, where it refers to the surrounding instance in which the code is executing. In all the cases that we've initialized libraries by passing in `this`, we've been handing the library a copy of the surrounding sketch. That's because all the code you write is implicitly running inside an instance of a class called `PApplet`, and some libraries need to be able to do things to the whole applet. The use of `this` also explains the magic behind initialization of `ControlP5` widgets. A method like `setPosition()` might look something like this internally:

```
class Button
{
  float my_x;
  float my_y;

  // Lots of other stuff...

  Button setPosition(
    float x, float y )
  {
    my_x = x;
    my_y = y;

    // Clever trick!
    return this;
  }
}
```

For the record, these are advanced programming tricks, and I wouldn't expect a student in this course to be able to use them. I simply felt obliged to explain a minor annoyance that has been popping up throughout the term.

There is one spot where it's occasionally useful to refer explicitly to `this`—in a constructor and in other methods that set local fields directly from arguments.

You can use the field name as the argument name and then clear up the ambiguity with `this`:

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

    Point( float x, float y )
    {
        this.x = x;
        this.y = y;
    }
}
```

As I said earlier, Object-Oriented programming is a huge topic. Like Human-Computer Interaction, many computer scientists spend their lives studying it, and there are whole research conferences devoted to the topic. Even within Java (and therefore in Processing) there's a lot more that's possible with OO—interfaces, inheritance, and templated types, just for starters. But the amount that we've seen here is enough to get started writing simple classes that simplify some programming tasks.

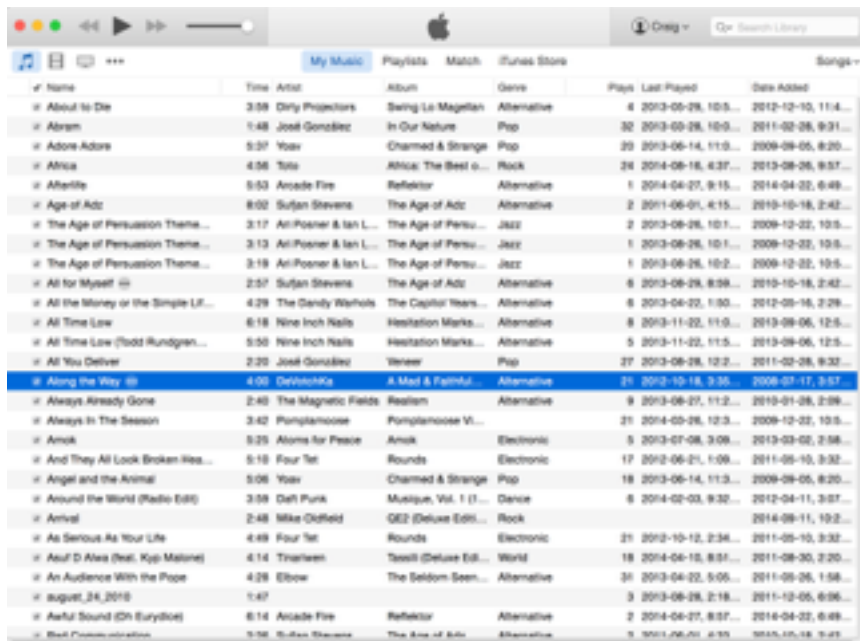


The shape of data

The programs we've written in this course don't typically manipulate large amounts of data. They tend to stick to a few small values, or at most a small collection of related objects. But in many real-world contexts, we need to maintain and manipulate large quantities of heterogeneous data (i.e., data made up of lots of different types, a mix of numbers, strings, booleans, and so on). I don't want to get too bogged down talking about concrete data structures and how to use them, but it's worth surveying the landscape briefly at a high level.

We've already talked a bit about dealing with arrays of instances. An array of instances looks something like a *database*. You don't have experience programming

with databases, but certainly with using them. Just about any large-scale web service (for starters, QUEST, JobMine, Twitter, Facebook, Amazon, etc., etc.) is backed by a database. For our purposes, we can think of a database as a large collection of records, each of which contains a similar arrangement of pieces of information. A good self-contained example of a database is your iTunes music collection:



Name	Time	Artist	Album	Genre	Plays	Last Played	Date Added
Abend in Die	3:38	Diny Projectors	Bring Lo Magellan	Alternative	4	2013-09-28, 10:5...	2012-12-10, 11:4...
Abram	1:48	José González	In Our Nature	Pop	32	2013-09-28, 10:5...	2011-02-26, 9:31...
Adore Adore	3:37	Yves	Charmed & Strange	Pop	20	2013-09-14, 11:0...	2009-09-05, 8:20...
Africa	4:58	Toto	Africa: The Best o...	Rock	24	2014-09-18, 4:37...	2013-09-26, 9:57...
Afraid	5:53	Arcaide Fire	Reflexor	Alternative	1	2014-04-27, 9:15...	2014-04-22, 6:49...
Age of Adz	8:02	Sufjan Stevens	The Age of Adz	Alternative	2	2011-06-01, 4:15...	2010-10-16, 2:42...
The Age of Persuasion Theme...	3:17	Art-Power & Ian L...	The Age of Persu...	Jazz	2	2013-09-28, 10:1...	2009-12-22, 10:5...
The Age of Persuasion Theme...	3:13	Art-Power & Ian L...	The Age of Persu...	Jazz	1	2013-09-28, 10:1...	2009-12-22, 10:5...
The Age of Persuasion Theme...	3:18	Art-Power & Ian L...	The Age of Persu...	Jazz	1	2013-09-28, 10:2...	2009-12-22, 10:5...
All for Myself	2:37	Sufjan Stevens	The Age of Adz	Alternative	8	2013-09-29, 8:58...	2010-10-16, 2:42...
All the Money or the Single L...	4:29	The Gendy Warhols	The Capital Years...	Alternative	8	2013-04-22, 1:50...	2012-09-16, 2:29...
All Time Low	6:18	Nine Inch Nails	Hesitation Marks...	Alternative	8	2013-11-22, 11:0...	2013-09-06, 12:5...
All Time Low (Todd Rundgren...	5:50	Nine Inch Nails	Hesitation Marks...	Alternative	5	2013-11-22, 11:5...	2013-09-06, 12:5...
All You Deliver	2:20	José González	Yveser	Pop	27	2013-09-28, 12:2...	2011-02-26, 9:32...
Along the Way	4:02	Delirious?	A Mad & Painful...	Alternative	21	2012-12-18, 3:30...	2008-07-17, 2:57...
Always Already Gone	2:40	The Magnetic Fields	Realism	Alternative	9	2013-09-27, 11:2...	2010-01-26, 2:08...
Always In The Season	3:42	Pumpkinseed	Pumpkinseed V...		21	2014-09-28, 12:3...	2009-12-22, 10:5...
Amok	5:25	Atoms for Peace	Amok	Electronic	5	2013-07-08, 3:09...	2013-03-02, 2:58...
And They All Look Broken Res...	5:15	Four Tet	Rounds	Electronic	17	2013-09-21, 1:09...	2011-05-10, 3:32...
Angel and the Animal	3:06	Yves	Charmed & Strange	Pop	18	2013-09-14, 11:3...	2009-09-05, 8:20...
Around the World (Radio Edit)	3:59	Daft Punk	Musique, Vol. 1 (I...	Dance	8	2014-02-03, 9:32...	2012-04-11, 3:07...
Arrival	2:48	Mike Oldfield	GE2 (Deluxe Edit...	Rock			2014-09-11, 10:2...
As Serious As Your Life	4:49	Four Tet	Rounds	Electronic	21	2013-10-12, 2:34...	2011-05-10, 3:32...
Asul D Aes (feat. Kyo Matome)	4:14	Tinariwen	Tassili (Deluxe Edit...	World	18	2014-04-15, 9:51...	2011-09-30, 2:20...
An Audience With the Pope	4:29	Elbow	The Seldom Seen...	Alternative	31	2013-04-22, 5:05...	2011-09-26, 1:58...
August 24, 2010	1:47				3	2013-09-28, 2:18...	2011-12-05, 6:06...
Aeolus Sound (DJ Eurydice)	6:14	Arcaide Fire	Reflexor	Alternative	2	2014-04-27, 9:57...	2014-04-22, 6:49...
Bad Communication	5:58	Sufjan Stevens	The Age of Adz	Alternative	5	2011-06-01, 4:15...	2010-10-16, 2:42...

Here, we might envision creating a `Song` class that holds all the information about an individual song: the title, the artist, the recording date, the duration, the genre, the rating the album name, the location of the file on your hard drive, the number of times it's been played, and so on. The entire database can then be viewed as an array of `Songs`.

In fact, this metaphor gives us another way to think about how we've been coding this term. Up until this module, we would have implemented this database by giving the *columns* priority: each column would have become a global variable, an array of the type of the information in that column. We would have to have worked hard to make sure all the columns stayed in sync with each other. Now, with objects, we can give the *rows* priority: we create a data type that can represent a single row of the database, at which point it's easy to make a single array of those rows.

There are two other standard “shapes” to collections of data that are worth mentioning:

- A *dictionary* is a way of associating a set of indexable *keys* to *values*. In the case of an actual dictionary, the keys are words and the values are their definitions. An old-fashioned dictionary is organized as a long list of key/value pairs. The keys are organized in alphabetical order to make the key you’re interested in easier to find (assuming your spelling is good!).

More generally, though, computer scientists think of a dictionary as any chunk of data in which you can look up the value attached to a particular key, add new key/value pairs, and remove a key and its value. We don’t need to think too hard about how this is accomplished; we simply trust the dictionary library to make sure these operations are efficient. A phone book is an example of a dictionary: the keys are names of people, and the values are their phone numbers. In this course, clicker registrations are a dictionary that maps clicker IDs to student IDs. The exam seating system is a dictionary that maps student IDs to seats in rooms. Dictionaries are used throughout computer programming.

- Many collections of data have some kind of hierarchical organization. The file system on your hard drive is organized as folders, where folders can contain files or other folders to any depth. A company’s org chart is another hierarchy, and has a simple structure as long as we assume that everybody has exactly one boss. Even an Illustrator document is hierarchical—the document is divided into layers, and each layer is made out of a nested hierarchy of groups. Groups can contain paths or other groups.

Hierarchical data like is usually organized into what computer scientists call a *tree*. Tree-structured data is also very important in computer programming. We’re not going to see a concrete example just yet, but hopefully by the end of the term we’ll have

encountered at least one tree in the form of the Processing class `JSONObject`.