CS 106 Winter 2016
Craig S. Kaplan

Module 01

Processing Recap

Topics

- **The basic "parts of speech" in a Processing program**
- **Scope**
- **Review of syntax for classes and objects**

Readings

- **Your CS 105 notes**
- ***Learning Processing*, Chapters 4–9**

Introduction

Let's review the main components of a Processing program. A quick review will help remind you of the ways that these pieces can fit together, and provide terminology that might reappear a few times during the term.

Make no mistake, creating an effective programming language is a *design problem.* That helps to explain why, as with the rest of the world of design, languages come in many styles, with different levels of popularity and fandom. If you're interested in thinking about the challenges in designing programming languages, I recommend Guy L. Steele's talk "Growing a Language".

Values

A *value* is any piece of information that can be manipulated by a running Processing program. Every program does what it does by creating values, moving them around, turning them into new values, etc. Simple values are things like numbers (`0`, `3.14159`) and booleans (`true`, `false`). A string (a block of text) is a more complicated value because we don't know how big it might be.

Some values can be seen as "compounds", bags of simpler values. An array is a value that holds a sequence of other values, which you can look up by an index. All objects, images, even the whole running sketch, are represented inside Processing as big complicated values.

Types

Every value in Processing belongs to a *type*, a collection of values that have similar behaviour and structure. For example, all whole numbers (`0`, `1`, `-4`, `99`...) belong to a type called `int`. Other important types in Processing include `float`, `boolean`, and `String`. The type `void` is special: it has no values, and is used to describe functions that don't give you back any information when you call them (`ellipse()`, for example).

If `X` is a type in Processing, then `X[]` is a new type "array-of-`X`". You don't even need to know anything about `X` in order to talk about arrays of them.

Expressions

An *expression* is any bit of code in a Processing program that produces a value when you execute it. We might also say that an expression "yields" a value.

The very simplest expressions are literally just literal descriptions of values, literally called *literals*. They produce themselves when executed—the text `37` is a very simple literal expression that yields the value `37`. A variable name like `width` is slightly more complicated. It looks as simple as a literal, but when the program runs we need to look up the value currently being held in that variable. And of course, a variable can hold different values at different times. That's why we need to distinguish between variable names and values.

There are many ways to combine simple expressions into more complicated ones. For example, any two expressions that describe numbers can be added together, giving a new expression that also describes a number. This ability to combine extends to lots of other operations in Processing: mathematical operators like `-`, `/`, and `*`, the square brackets used to look things up in arrays, the syntax for calling functions, and so on. The ability to nest expressions arbitrarily inside of other expressions like this is a Big Idea in programming.

Here's a handy rule of thumb: if you can put a piece of code inside of a `println()`, then that code is an expression.

Statements

A *statement* is a bit of code that *does something*, that alters the internal state of a running program. When I write a line like

```
a = 5;
```

my goal is to permanently change the program by throwing out whatever value is currently stored in the variable `a`, and putting the value `5` in its place. If I ask for an ellipse to be drawn with

```
ellipse( 50, 50, 100, 100 );
```

I'm not expecting the function call to give me an answer; I want Processing to do something to the screen.

Like expressions, some statements can be more complicated structures made out of smaller pieces. Most obviously, `if`, `while` and `for` are all keywords that introduce compound statements.

Declarations

A *declaration* is a piece of code that introduces a new name into the program. The simplest declarations create new variables. Thus I can write

```
int a;
float b = 3.14;
```

and from that point onward (at least until the variables go out of scope!), the names `a` and `b` mean something in the program. The second declaration above also includes an initialization expression.

Of course, it's also possible to declare functions:

```
float distance(
  float x1, float y1, float x2, float y2 )
{
  return sqrt( sq( x1 - x2 ) + sq( y1 - y2 ) );
}
```

This declaration makes the name `distance` available for the rest of the program. It also introduces four parameter names `x1`, `y1`, `x2` and `y2`, which are visible only inside the body of the function.

The most complicated declarations are *class declarations*. They introduce a new type into a Processing program. Inside that class, we may declare an arbitrary set of new *fields* and *methods* (which are the inside-a-class analogues of variables and functions). A simple class declaration might look like this:

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

  Point( float xx, float yy )
  {
    // ...
  }

  float magnitude()
  {
    // ...
  }
}
```

Here we introduce a new type called `Point`. Every `Point` has fields `x` and `y`, a method `magnitude()`, and a special method called a *constructor* (whose name is the same as the name of the type). Class declarations are the basis for object-oriented programming.

Scope

Every declaration is confined to live within a definite *scope*, a range of the text of the program where that declaration is visible and can be referred to. For example, if you declare a variable at the "top level" of a sketch, it's visible everywhere in the sketch, unless some other scope inside the sketch decides to define its own variable of the same name. On the other hand, a variable declared inside of a function is only visible in the body of that function, and doesn't have meaning to the outside world.

Scopes can be nested inside of other scopes. As a rule of thumb, just about any time you see an open curly brace (a `{` character), it represents the start of a new scope in which you can declare new variables. That means that the bodies of functions and methods are nested scopes, as are the insides of blocks of code in `if` statements and `for` and `while` loops.

We're not going to spend too much time this term worrying about scope. But it's important to be aware of the concept, since misunderstanding of scopes can be the cause of some programming errors. See Section 6.5 of *Learning Processing* for a few more details about scope.

Program

A *program* is a sequence of declarations. That's it! The only thing you're allowed to do at the top level in a Processing program (assuming we're not using "immediate mode") is to declare things. If that's the case, how does any work ever get done? Well, Processing expects you to write functions with certain agreed-upon names—most obviously, `setup()` and `draw()`. If you write those functions, Processing promises to call them at appropriate times. We sometimes refer to such functions as "hooks".

Classes and objects

Working with classes requires the use of a few special bits of syntax. Let's review them briefly (review Chapter 8 of *Learning Processing* for a more in-depth discussion of classes).

We use the special keyword `new` to create objects:

```
Point p = new Point( 3, 4 );
```

Immediately after `new`, you write the name of the class you want to create an instance (i.e., a new object) of. Then you put some arguments in parentheses, as you would if you were calling a function. These arguments are passed to the class's constructor.

Once we have an instance, we can use "dot notation" to reach inside that instance, look at its fields and call its methods.

```
println( p.x );
p.x = q.y;

float m = p.magnitude();
```

On the left of the dot, we write any expression that names an instance of a class. It could be something simple like a variable name, as above, or a more complicated expression like looking up an instance in an array. On the right we put the name of a field or method.

There are two special names to be aware of in the context of objects and classes:

- `null` is a special value that belongs to every class, meaning "no legal object". It's the default value for variables of class type, and it's distinct from every possible legal instance of that class. (See Page 146 of *Learning Processing.)*
- Inside of any method of any class, you're allowed to use the magic keyword `this`, which means "the instance that this method was called on". In the code `p.magnitude()` above, we can think of "sending the `magnitude` message to the object referred to by `p`". Then, inside the body of the `magnitude()` method, the name `this` will refer to *that object.* Sorry, what? Yes, this is confusing, and *Learning Processing* avoids mentioning it until Page 330. It's an ugly wart in Processing, a place where they expose the messy Java infrastructure that powers

Processing in a way that would best be left hidden. We won't ever have to write our own code that does real work with `this`, but when we start working with external libraries we'll occasionally have to refer to it.