Classes #3

These notes discuss the class code cc16. You should have that code in the Class1 folder open as you read through this.
This third set of notes has two goals:

1. correct an intentional error in the code presented in the previous two sets of notes and class code
2. briefly summarize the use of classes

First, the error. From the previous set of class code, the code below has the error marked in red

```
// Class Code Set 16
// Classes in Processing  #1

Square s;

void setup( )
{
   size( 600, 600 );

   s = new Square( );
   s.setup( );

   background( 0 );
}

void draw( )
{
   s.move( );
   s.draw( );
}
```

```
class Square
{
  // fields or variables
  float x, y, edge, dX, dY;
  color col;

  void setup( )
  {
     x = random( 10, 500 );
     y = random( 10, 500 );
     edge = random( 50, 200 );
     dX = random( 2, 10 );
     dY = random( 2, 10 );
     col = color( random( 255),
                  random( 255),
                  random( 255 ) );
  }
  void move( )
  {
     . . .
  }
  void draw( )
  {
    . . .
  }
}
```

Jim used the setup( ) function so the structure of this code would parallel the code you have been writing since week #2. However, this is not the "correct" or "proper" way to initialize the variables in the object. Here is the "correct" or "proper" code.

```
// Class Code Set 16
// Classes in Processing  #3

Square s;

void setup( )
{
   size( 600, 600 );

   s = new Square( );

   background( 0 );
}

void draw( )
{
   s.move( );
   s.draw( );
}
```

```
class Square
{
  // fields or variables
  float x, y, edge, dX, dY;
  color col;

  Square( )
  {
     x = random( 10, 500 );
     y = random( 10, 500 );
     edge = random( 50, 200 );
     dX = random( 2, 10 );
     dY = random( 2, 10 );
     col = color( random( 255),
                  random( 255),
                  random( 255 ) );
  }
  void move( )
  {
     . . .
  }
  void draw( )
  {
    . . .
  }
}
```

There are two changes:

1. The call of the setup( ) function for the Square reference, s on the left side is gone
2. The first line of the original setup() function on the right side in the definition of the Square class has been seriously modified.

The setup() function definition has been replaced with a Square() function definition – sorta'. . . The code is the same – only the name has been changed. If we look carefully at the Square( ) definition, we see what looks like some errors:

1. There is no return type
2. The function has the same name as the class
3. In the left column we see that it is called in a very different way.

HOWEVER, this runs properly. Processing hands this code to the Java compiler and we see the square moving around the window.

So what is going on?

The code that defines Square( ) is different because Square( ) is a special kind of function [1] called a constructor. The constructor's purpose is to initialize the variables to their correct starting values.

The rules for defining and using a constructor are:

1. there is no return type,
2. the name of the constructor MUST be exactly the same as the name of the class, and
3. it can only be used after the new operator

We can use parameter/argument binding for constructors just as we use them for functions. Constructors have signatures just like functions so we can define multiple constructors as long as they have different signatures.

What about inheritance and constructors?

Nothing said above contradicts what we have done with OOP. Look at the code in the folder Class2. The only differences between this code and the code in the Class2 folder in code set 15 are:

1. the setup() function is now the shape( ) constructor
2. the call of the setup( ) function in the initArray( ) function is gone

Summarizing Classes and OOP

This whirlwind look at Classes is far from complete in either concepts or syntax. There is much more. What you have here is enough to get you started with classes and to use them in your remaining work. The "real" study of OOP in a "real" programming class goes much deeper with more complex syntax. This is because the programs that typically use OOP are concerned with keeping the data correct and not allowing it to be corrupted by functions outside of the class. If we are writing code for a bank or an investment firm, this is important. BUT we are not doing that. We are using OOP to simplify our coding for the art we want to generate. The data is ours and rarely if ever will a function outside of our control work with the data. This is one reason we can do a very lite look at OOP and take advantage of what it offers.

---

[1] Some authors insist that the constructor is not a function. It is a constructor. They also insist that constructors are not called. So the line in the code in the left box:

```
s = new Square( );
```

is not a function call (according to them).

Here is a brief summary of some of the important ideas in these last few days of work.

The idea of Object Oriented Programming (OOP) is to capture, or encapsulate, or code everything we need to represent or model something in one programming entity that we call a class. Doing this is called Object Oriented Programming or OOP. In these examples we used the classes Square and Circle.

Depending on the problem we must solve, we could define a class to help us. Suppose we were writing a new card game. We could define a class for one playing Card . Then we could define a class for a Deck that would have an array of references to 52 Card objects. Each Card object would have its variables assigned the values of one particular card.

Or we could write an adventure game where the player gathers "stuff" to help her/him. This stuff could be magic potions, weapons, spells, armor, etc. We could use classes to help us write the code. We might define a Stuff class as a parent class and use inheritance to define the individual stuff the player could collect.

Done well, Object Oriented Programming offers some advantages over a single large program with many functions:

- a method for breaking the problem into sub-problems
- easier and quicker debugging
- easier and quicker modification

Since a good class breakdown can allow for some of the classes to be coded and tested separately from the rest of the code, development can proceed in parallel for much of the work.

To use classes in our code we need different "kinds" of files:

- a "client" file that "hires" a class to provide a needed service.
- a "server" file that provides the needed service. These are the classes that we have just worked with. They have the variables and functions to do what is needed by the client.

In our work over the last few days, the client was the
same the file we have been writing for 11 weeks. It has the setup() and draw() functions as well as any other functions need.
These files declared objects of the classes (references to the classes) - the servers, newd them, and called their functions.

The classes we wrote(Shape, Square, Circle) were the server files. They were the classes that defined the Square, Circle, and Shape.

Classes have two basis types of stuff: variables and functions. These are often given different names:

- variables can be called instance variables , fields, and state
- functions are often called methods, member functions, and behavior

The setup() function is replaced with a constructor that has the following rules for its coding and use:

1. There is no return type
2. The name of the constructor MUST be exactly the same as the name of the class
3. It can only be used after the new operator

Classes define what Processing must do to build and use objects of the class.

To use the class, an object of the class must be built. This idea is very similar to building an array. In both cases we start with references but both references are null;

```
int  [ ] a;
Square s;
```

The next task is to create the actual object that has the data we need. Again, this is similar to how we create the actual array:

```
a = new int [ 5 ];
s = new Square( );
```

When these two lines of code are executed, we can use this terminology to describe what has happened:

"s references a Square object"

just as we say,

" a references an array of int "

To use the data in the object or call its functions <u>from the client</u>, we use the dot or period syntax that shows possession:

```
s.move( );
if ( s.x > width ) . . .
```

If we have two or more very similar classes to define, we can take advantage of inheritance. To do this, we define a parent class that has the data and functions that are common to the similar classes. Then we can code the similar class as sub-classes or extended classes so they inherit the common "stuff". In the individual sub-classes we define only those functions that are different from the other sub-classes.