02 Encapsulation

Object Oriented Programming

Program Design – complex programs must be broken down into parts; each part is easier to program by itself; then the parts can be reassembled into a complete program.

Coupling – the degree to which each part depends on the others. For example one function may call many other functions (high coupling) or it may only depend on one or two other functions (low coupling).

OO Design – a type of program design in which the parts are *objects*, e.g a university is composed of students and lecturers; a slideshow is composed of many images.

Object – an entity in your program that has properties (fields) and behaviours (methods). Often corresponds to something in the real world (e.g. a slide, a student).

Class – a template or "cookie cutter" for objects. A class is the *type* of an object just as `int` is the type of a variable.

Built-In Classes – Processing has many useful classes that you can use in your programs such as `PVector`, `PImage`, `PFont` etc.

Examples

	"Real world" Examples		Processing Examples		
	Lecturer	Planet	PVector	PImage	PFont
Fields	name, address, classes taught	name, mass, radius, length of year	x,y,z	pixel data, image width/height	ttf data, font size, ascents etc
Methods	gives a class, sleeps, walks the dog	rotates, orbits a star, may get warmer etc	can be rotated, multiplied etc	can be, resized, filtered (e.g. turned into grey), cropped etc	can be saved to a file

Built-In Classes – check the Processing reference to see what all the fields/ methods are for classes, e.g. http://processing.org/reference/PImage.html

Object References – the name of an object is also called a reference. The following example creates some null object references:

```
PImage myImage;
Student someStudent;
```

Object creation – use either the `new` keyword or one of Processing's handy create or load functions. Note that *an object reference is not the same thing as an object*.

```
PImage myImage = loadImage("dog.jpg");
Student someStudent= new Student("Smith", "John",
                     "1722772");
```

The object reference is a "pointer" to the location of an object in memory.

Objects with many references – one object may have as many object references as you like. This is a point of difference between objects and primitive data types.

```
// Create three references but only one object
PFont myFont = createFont("Sans-Serif", 56);
PFont anotherReference = myFont;
PFont yetAnotherReference = myFont;

// Create three characters
char someCharacter = 'x';
char anotherCharacter = someCharacter;
char yetAnotherCharacter = someCharacter;
```

Null – object references that do not reference any object have a value of `null`.

```
PImage myImage = loadImage("dog.jpg");
/* … do something with the image */
myImage = null; // effectively deletes the image
```

Accessing Fields – once an object is created, use dot (".") notation to access the fields, e.g. `println(backgroundImage.width);`

Methods – a method is a function that is attached to an object. Methods are sometimes also called *messages*, e.g. tell the dog (the object) to bark (the message).

Calling Methods – methods are also called using dot notation, e.g.

```
position.mult(5);
theDog.bark();
```

Passing By Reference

Primitive data types -- always *copied* by assignment, e.g.

```
int x=3;
int y=x;
y++;
println(x);
println(y);
```

outputs

```
3
4
```

Passing by value – parameters are always *copied* to functions, e.g.

```
void increment(int x) {
  x++;
  println(x);
}
…
int a=1;
increment(a);
println(a);
```

outputs

```
2
1
```

Copying/Passing by reference – objects are *not* copied by assignment or function calls. Instead, only their *references* are copied.

Common Mistake:

```
PImage fireImage, fireImageProcessed;
void setup() {
  fireImage=loadImage("fire.jpg");
  fireImage.resize(300, 0);
  fireImageProcessed=processImage(fireImage);
}

PImage processImage(PImage image) {
   image.filter(GRAY);
   image.filter(BLUR);
   return image;
}
```

If you want to copy objects, in general you need to do it yourself:

```
PImage processImage(PImage image) {
  int w=image.width, h=image.height;
  PImage result = createImage(w, h, image.format);
  result.copy(image, 0, 0, w, h, 0, 0, w, h);
  result.filter(GRAY);
  result.filter(BLUR);
  return result;
}
```

More Object Oriented Programming Terms

Enscapsulation –refers to the way that data and behaviours are wrapped together inside objects, e.g. all 2D and 3D vector data and code are wrapped together in the `PVector` class.

Data Hiding – not all data inside an object needs to be visible. Data that is not meant to be read/writable outside of an object should be hidden, e.g. a bank account balance.

Programming a Class in Processing

Components – class name (e.g. `Account`), fields (e.g. `balance`, `name`, `accountID`), methods (e.g. `withdraw()`).

Fields – should be divided into *public* and *private*

Methods – may also be *public* or *private*

Example:

```
class Account {

  public String accountName;
  public String accountID;
  private float accountBalance=0;  // Not accessible!!

  public float getAccountBalance() {
    return accountBalance;
  }

  public void deposit(float amount) {
    accountBalance += amount;
  }

  public void withdraw(float amount) {
    if (accountBalance-amount>=0)
      accountBalance-=amount;
  }
}
```

Object creation – once a class is defined, it can be used to create as many objects as you like. You must declare a reference for each object that you want, e.g.

```
annAccount = new Account();
johnAccount = new Account();
```

Public field usage -- Outside of the class, public fields are accessible, e.g

```
annAccount.accountID = "#2239738383";
```

Public method usage – similarly, all public methods are available outside of the class, e.g.

```
johnAccount.deposit(5);
```

Constructors

Constructors – used to set up default values for fields when new objects are created, for example suppose new space invaders should (i) be created at a random position on the screen with a certain velocity and (ii) have a custom image loaded to represent them:

```
class Invader {

   private PVector position, velocity;
   private PImage image;

   public Invader(PImage sprite) {
      image = sprite;
      float x = random(width);
      float y = random(height);
      position = new PVector(x,y)
      velocity = new PVector(1, 0);
   }

/* … rest of class … */

}
```

Constructor invocation – constructor executes whenever the object is created with new, e.g.

```
PImage mySprite = loadImage("invader.png");
Invader someInvader = new Invader(mySprite);
```

Object Oriented Design

OOD -- the problem of deciding what the most appropriate classes are for your program, and how the objects will interact. You usually get given a requirements document or a brief to start with.

Nouns in the brief – usually a good idea for classes, e.g. Alien and Player in a Space Invaders game.

Verbs in the brief – usually a good idea for the methods, e.g. shoot or move.

Adjectives in the brief – usually a good idea for the properties, e.g. score, position, health.

Example: A virtual art viewer is a program designed to allow users to browse digitally scanned works of art. Each work of art will have a history, artist and other details associated with it that users can read. The user will be able to zoom in and explore the work of art in detail, and keep their own notes about each work. Users will also be able to keep albums of their favourite works for others to share.

Collections

Collections – refer to a group of objects of the same class. These are frequently useful for managing the objects in your program. The most common is the `ArrayList`.

```
// Reference to a collection of invaders
ArrayList<Invader> invaders;

void setup() {
  // Preload the sprite for the invader
  PImage invaderSprite = loadImage("invader.png");

  // Create the collection of invaders
  invaders = new ArrayList<Invader>();

  // Create the individual invaders
  for (int index=0; index<NUM_ALIENS; index++) {
    Invader invader = new Invader(invaderSprite);
    invaders.add( invader );
  }
}
```

ArrayList for loops – ArrayLists allow a special form of the for loop that does not require an index variable:

```
void draw() {
  // Draw the invaders
  for (Invader invader: invaders)
    invader.draw();

  // Move the invaders
  for (Invader invader: invaders)
    invader.move();
}
```

Safe Access to Fields

Public fields – take care with these, as public field access is uncontrolled. This causes many problems for new programmers. For example:

```
Class Invader {
     public PVector position;

     /* … rest of class … */
}
```

allows these obviously incorrect statements:

```
myInvader.position.x = -100000;     // unintended x value
myInvader.position=null;            // will cause a crash
```

Get/Set methods – use these to enable safe read-only or read/write access to fields, e.g.

```
Class Invader {
     private PVector position;
     /* … rest of fields … */

     public PVector getPosition() {
          return position.get();   // copies the PVector
     }

     public PVector setPosition(float x, float y) {
          position.x=x;
          position.y=y;
     }

     /* … rest of class … */
}
```