# 02 Encapsulation

## Object Oriented Programming

|  |  |
| --- | --- |
|  |  |

**Encapsulation** – A way of packing data and functions into a single component. Example: The Sims character has state (hungry, sleepy, etc) and functionality (e.g. tell the Sim to start eating). A projector has state (volume, what is plugged in, which buttons are on, etc) and functionality (e.g. press one of the buttons).

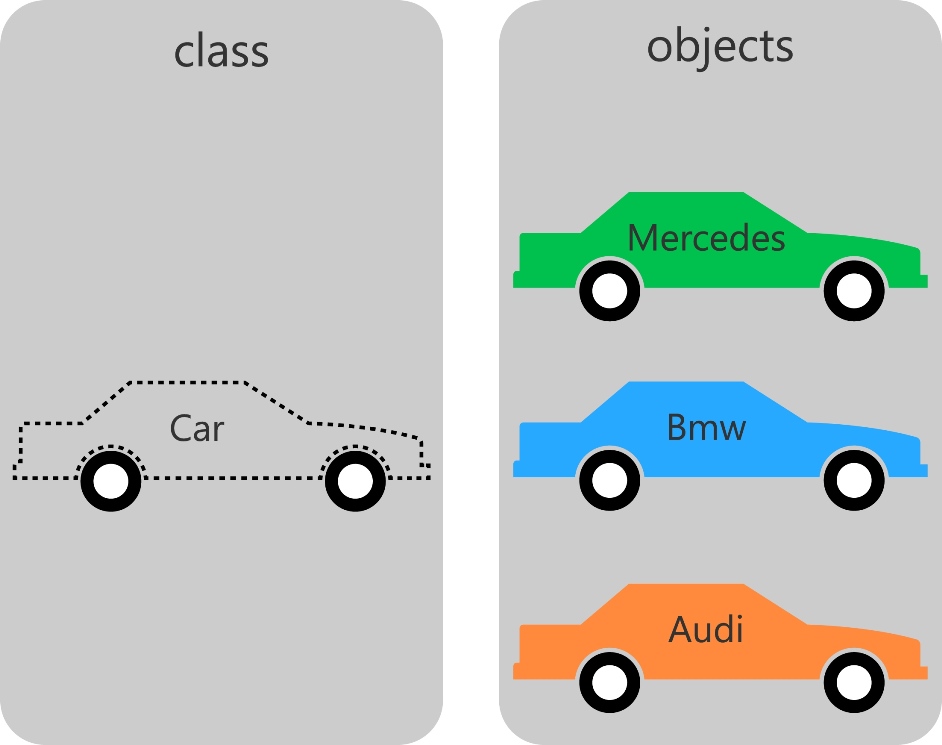
*The exact way in which the object’s state changes in response to its functions being invoked (which might be complicated) is usually hidden from the user (by a simple interface).*

**Object Oriented Programming** – a type of programming in which a program is made of components called **objects**. Objects have their own *states* and *behaviours* and can *interact*.

An object’s **state** is also referred to a **property** or a **field**.

An object’s **functionality** is also referred to as its **methods**.

**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. By itself a class is not much use; it is only useful when you start creating objects from it.



**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 called a *reference*. The following example creates object references:

PImage myImage;

PVector startPoint,endPoint;

**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*. A reference is a like a name: one object can have many of them just as one object in the real world can have many names (e.g. “John”, “Mr Smith”, “Your Majesty” etc may all refer to the same person).

PImage myImage;

PVector startPoint,endPoint;

myImage = loadImage(“dog.jpg”);

startPoint = new PVector(13.5,-23);

In the above example, endPoint is a reference that has not been initialized.

**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);

**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;

}

## 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 invader 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) {

// optionally check that x and y are correct

// then…

position.x=x;

position.y=y;

}

/\* … rest of class … \*/

}