Introduction à la Programmation Orientée Objets

# Introduction

## Objectif du cours

A travers le développement d'une ébauche de jeu de bataille navale, l'objectif de ce cours est :

* de comprendre comment on passe de la programmation structurée à la programmation orientée objets ;
* de comprendre l'intérêt et le principe de l'encapsulation des données en programmation orientée objet ,
* de découvrir le langage de programmation Haxe et la bibliothèque OpenFL.

Le développement s'effectue en langage Haxe qui bien que peu connu (mais très similaire au C# et au Java) à l'avantage de la simplicité et de permettre avec un seul langage de développer des applications Web, des jeux videos (multi plate-forme), … L'objectif de l'enseignement est transversal et consiste :

* à étudier les concepts de la Programmation Orientée Objets (POO),
* à maîtriser un environnement de développement intégré (IDE),
* à apprendre à se repérer dans une documentation.

## Préparation

Mettre en place l'arborescence suivante :

SLAM2

└── Introduction //chaque TP aura son propre dossier

├── 1-ProgStructuree

├── 2-ProgObjet

├── 3-Recap1

├── 4-Collections

└── 5-Recap2

# Étape n°1 : de la programmation structurée à l'objet

## Solution structurée (sans POO)

* Ouvrir un terminal dans le dossier 1-ProgStructuree.
* Initialiser un nouveau projet : openfl create project BatailleNavale ; puis supprimer le fichier hxproj (utile uniquement pour l'IDE FlashDevelop)
* Placer les images mer.png et navire.png dans le dossier Assets (ressources).
* Éditer le fichier Main.hx (dossier Source) et le compléter comme indiqué ci-après.
  + Supprimer l'instruction package; les paquetages servent à organiser le code en dossier et sous-dossiers ; par exemple, le code de la classe Sprite est dans le dossier display du dossier openfl de la bibliothèque (/usr/lib/haxe/extra/openfl/*N°Version*/openfl/display/Sprite.hx).
  + Importer les classes (types) nécessaires :

import openfl.Assets;

import openfl.display.Bitmap;

import openfl.display.BitmapData;

* + Importer une classe permet d'éviter de devoir indiquer son chemin à chaque utilisation ; par exemple, si la classe Sprite n'était pas importée, il faudrait écrire :   
    class Main extends openfl.display.Sprite {
  + Entre les import et la classe **Main**, définir un type Navire :

typedef Navire = {

var x : Int;

var y : Int;

var bmp : Bitmap;

var mer : Sprite;

}

*Un navire a une position, une image et navigue sur une mer.*

* + Dans la classe **Main**, ajouter la fonction permettant de dessiner la mer :

function **dessinerMer**() : Sprite {

var mer : Sprite = new Sprite();

var pixels : BitmapData = Assets.getBitmapData("assets/mer.png"); //chargement des //données de l'image

for (i in 0...15) { //pour chaque ligne

for (j in 0...15) { //pour chaque colonne – cf for (j=0; j<15; j++)

var bmp : Bitmap = new Bitmap(pixels); //création d'une image

mer.addChild(bmp); //ajout (affichage) de l'image sur le « calque »

bmp.x = i \* 32; //positionnement de l'image (de taille 32x32)

bmp.y = j \* 32;

}

}

return mer;

}

*En programmation en deux dimensions les calques (sprites) servent à regrouper les objets graphiques ; ici la mer est constituée d'une mosaïque de 15 x 15 images (bitmaps).*

* + Dans la classe Main, ajouter la fonction permettant de dessiner un navire :

public function **dessinerNavire**(nav : Navire) {

nav.bmp = new Bitmap(Assets.getBitmapData("assets/navire.png"));

nav.mer.addChild(nav.bmp); //affichage du bateau sur la mer

nav.bmp.x = nav.x \* 32; //positionnement de l'image (de taille 32x32)

nav.bmp.y = nav.y \* 32;

}

* + Compléter la fonction new (qui initialise le programme) – après l'instruction super() :

var laMer : Sprite = dessinerMer();

addChild(laMer); //ajout de la mer au calque qui recouvre la scène

laMer.x = 160; //centrage: (800-15\*32)/2

laMer.y = 60;

var n1 : Navire = { x : 5, y : 5, mer : laMer, bmp : null }; //pas d'image pour l'instant

var n2 : Navire = { x : 10, y : 10, mer : laMer, bmp : null };

dessinerNavire(n1);

dessinerNavire(n2);

*Avec OpenFL, le jeu est un calque (recouvrant la scène) sur lequel on positionne les autres objets.*

* Exécuter le programme : depuis le dossier BatailleNavale, saisir la commande : openfl test neko
* Ajouter un autre navire à la position (16,16).

## Solution avec Programmation Orientée Objets

* Ouvrir un terminal dans le dossier 2-ProgObjet.
* Initialiser un nouveau projet : openfl create project BatailleNavale.
* Créer un nouveau fichier Navire.hx (dans le dossier Source) avec le contenu suivant :

import openfl.display.Bitmap;

import openfl.display.Sprite;

import openfl.Assets;

class Navire {

private var x : Int;

private var y : Int;

private var mer : Sprite;

private var bmp : Bitmap;

public function new(x : Int, y : Int, laMer : Sprite) {

if (x>=0 && x<15) this.x = x;

else this.x = 0;

if (y>=0 && y<15) this.y = y;

else this.y = 0;

this.mer = laMer;

}

public function dessiner() {

this.bmp = new Bitmap(Assets.getBitmapData("assets/navire.png"));

this.mer.addChild(this.bmp);

this.bmp.x = this.x \* 32;

this.bmp.y = this.y \* 32;

}

}

*L'identificateur this permet de désigner le navire qu'on est en train de manipuler (créer ou dessiner).*

* Éditer le fichier Main.hx (dossier Source) et le compléter comme indiqué ci-après.
  + Récupérer la fonction dessinerMer du programme précédent.
  + Remplacer les dernières instructions de la fonction new par les suivantes :

var n1 : Navire = new Navire(5, 5, laMer);

var n2 : Navire = new Navire(10, 10, laMer);

n1.dessiner();

n2.dessiner();

* Tester le programme.
* Ajouter les instructions suivantes (et tester) :

var n3 : Navire = new Navire(16, 16, laMer);

n3.dessine();

* Ajouter l'instruction (à placer entre var n2 : Navire … et n2.dessine()) :

n2.x = 5;

*Le programme ne peut être compilé, car la propriété x du navire est privée (inaccessible en dehors de sa classe) – c'est le principe de l'encapsulation pour protéger les données et rendre les programmes plus robustes (les utilisateurs de la classe sont « verrouilles »).*

## Synthèse

### Questions de réflexion

Observer les différences entre les deux programmes :

* entre la classe et le type Navire ;
* entre la fonction dessinerNavire et fonction (méthode) dessiner de la classe Navire.

Expliquer comment une fonction peut accéder aux propriétés d'un objet.

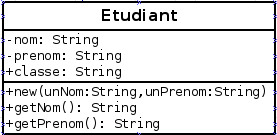
Expliquer comment la programmation orientée objet permet d'améliorer la robustesse des programmes.

### Concepts abordés

* Structure de données.
* Classe, propriété, méthode, objet / instance.
* Visibilité publique / privée et protection (encapsulation) des données.
* Allocation mémoire, constructeur.

# Exemple récapitulatif

La classe Etudiant définit les caractéristiques d'un étudiant et ce qu'il est possible de faire avec un objet de cette classe. Le diagramme de classe UML (Unified Modeling Language) suivant en décrit les propriétés et méthodes ; remarquer que la propriété classe est publique (donc non protégée) ; le constructeur de la classe prend deux paramètres : le nom et le prénom de l'étudiant à créer.

* Repérer dans le programme suivant les instructions « interdites » et en expliquer la raison :

class Main {

public static function main() {

var e : Etudiant = new Etudiant("Asimov", "Isaac");

trace(e.nom);

trace(getNom(e));

trace(e.classe);

}

}

* Dans le dossier 3-Recap1, coder la classe Etudiant, puis compléter le fichier Tests.hx (remplacer les lignes en gras par les actions décrites en commentaire) :

import haxe.unit.TestCase;

import haxe.unit.TestRunner;

class TestEtudiant extends TestCase {

function testNew() {

**//déclarer la variable "e" comme étant de type "Etudiant"**

**//créer un nouvel étudiant "Bordage Pierre" (à affecter à la variable "e")**

**//recupérer le nom de l'étudiant**

assertEquals("Bordage", nom); //vérifie si le nom est correct

}

function testClasse() {

**//déclarer une variable "e" et instancier un nouvel étudiant**

**//affecter la classe "sio1" à cet étudiant**

assertEquals("sio1", e.classe);

}

}

class Tests {

public static function main() {

var r : TestRunner = new TestRunner();

r.add(new TestEtudiant());

r.run();

}

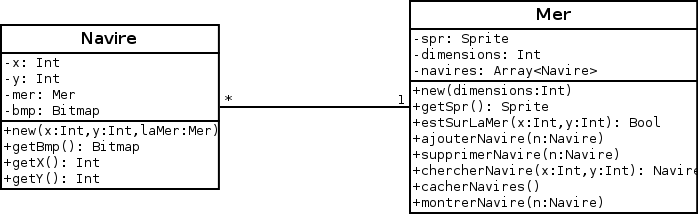
}

* Compiler le programme (OpenFL n'est pas utilisé) : haxe -main Tests -neko tests.n
* Tester le programme avec la commande : neko tests.n

# Étape n°2 : UML et collections (tableaux)

## Exemple concret

L'exemple de la bataille navale est repris et complété avec une nouvelle classe « métier » Mer. Une mer est constituée d'un calque (spr) ayant un nombre de lignes et colonnes (cf dimensions) et a des navires. Les propriétés (également appelées attributs) et méthodes publiques sont précédées de « **+** », celles privées de « **-** ».

* Créer avec le logiciel Dia ce diagramme UML, et l'enregistrer dans le dossier 4-Collections.
* Créer un nouveau projet OpenFL dans le dossier 4-Collections.
* Générer ensuite le code des classes métier en effectuant un export au format XSL/code (UML→Haxe). Remarque : si le langage Haxe n'est pas disponible, [installer « l'extension »](http://sallu.tuxfamily.org/-Haxe-utilities-) et redémarrer Dia.
* Compléter le code des classes Navire et Mer dont le corps de certaines méthodes sont fournies :

public function **new**(dimensions : Int) {

this.dimensions = dimensions;

this.navires = new Array();

this.spr = new Sprite();

var image : BitmapData = Assets.getBitmapData("assets/mer.png");

for (i in 0...dimensions) {

for (j in 0...dimensions) {

var bmp : Bitmap = new Bitmap(image);

this.spr.addChild(bmp);

bmp.x = i \* Main.TAILLE;

bmp.y = j\* Main.TAILLE;

}

}

}

public function **ajouterNavire**(n : Navire) {

this.navires.push(n);

this.spr.addChild(n.getBmp());

}

public function **supprimerNavire**(n : Navire) {

this.navires.remove(n);

this.spr.removeChild(n.getBmp());

}

public function **chercherNavire**(x : Int, y : Int) : Navire {

var nc : Navire = null;

for (n in navires) {

if (n.getX() == x && n.getY() == y) {

nc = n;

break; //met fin à la boucle

}

}

return nc;

}

public function **cacherNavires**() {

for (n in navires) {

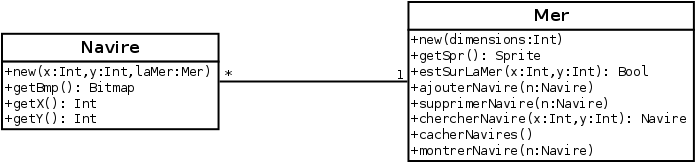
n.getBmp().visible = false;

}

}

*La propriété navires de la classe Mer est un tableau (Array) permettant de mémoriser plusieurs éléments du même type (une collection). Généralement, un tableau est initialisé dans le constructeur ; les méthodes push et remove permettent ensuite respectivement de lui ajouter ou retirer des éléments.*

Pour permettre à un autre développeur d'utiliser les classes Mer et Navire, l'auteur de ces dernières peut communiquer le diagramme UML suivant sur lequel les propriétés et méthodes privées n'apparaissent plus :

* Compléter le code de la classe Main :

import openfl.events.MouseEvent;

import openfl.display.Sprite;

class Main extends Sprite {

public static inline var TAILLE : Int = 32; //static inline → constante

public static inline var NBNAV : Int = 4;

var mer : Mer;

var nbNav : Int;

public function new() {

super ();

nbNav = NBNAV;

mer = new Mer(15);

addChild(mer.getSpr());

mer.getSpr().x = (800 - 15 \* TAILLE) / 2;

mer.getSpr().y = (600 - 15 \* TAILLE) / 2;

mer.getSpr().addEventListener(MouseEvent.CLICK, onMerClick); //définit la   
 //fonction à exécuter en cas de clic sur la mer

}

public function **onMerClick**(e : MouseEvent) {

var x : Int = Math.floor(e.localX / TAILLE);

var y : Int = Math.floor(e.localY / TAILLE);

if (nbNav > 0) { //de NBNAV à 1

var n : Navire = new Navire(x, y, mer);

**//ajouter le navire à la mer**

nbNav--;

} else if (nbNav == 0) {

**//cacher les navires**

nbNav--;

} else if (nbNav >= -NBNAV) { //de -1 à -NBNAV

var n : Navire = mer.chercherNavire(x, y);

if (n != null) {

mer.montrerNavire(n);

nbNav--;

}

} else { //nbNav < -NBNAV → on recommence

nbNav = NBNAV;

//supprimerNavires();

}

}

}

* Ajouter la méthode supprimerNavires à la classe Mer : elle doit d'abord retirer les images de chaque navire, puis réinitialiser le tableau.

## Synthèse

Cet exemple a abordé :

* l'ébauche de **diagramme des classes UML** (Unified Modeling Language) : classes, propriétés, méthodes, visibilité publique / privée, associations et multiplicité ;
* la notion de «**navigation**» entre objets de classes différentes et celle de «**collection**» (implémentées sous la forme de tableaux) : initialisation, ajout et retrait d'éléments.

# Exemple récapitulatif

* Créer le diagramme UML des classes décrivant les classes Etudiant (cf exemple récapitulatif n°1) et Section (« tstmg1 », « sio1 », « sio2 », …) ; une section à une propriété (privée) etudiants (un tableau). L'analyse du programme de test suivant (fichier Tests.hx du dossier 5-Recap2) doit permettre de déterminer les méthodes de la classe Section  :

import haxe.unit.TestCase;

import haxe.unit.TestRunner;

class TestSection extends TestCase {

function testNew() {

var s : Section = new Section();

assertEquals(0, s.getNbEtudiants());

}

function testAjoutEtudiant() {

var s : Section = new Section();

var e1, e2 : Etudiant;

e1 = new Etudiant("Barjavel", "René");

s.ajoutEtudiant(e1);

assertEquals(1, s.getNbEtudiants());

e2 = s.getEtudiant(0);

assertEquals("René", e2.getPrenom());

}

}

class Tests {

public static function main() {

var r : TestRunner = new TestRunner();

r.add(new TestSection());

r.run();

}

}

* Copier le fichier Etudiant.hx du premier récapitulatif dans le dossier 5-Recap2. Créer ensuite la classe Section de sorte que le programme suivant réussisse les tests :

# Pour aller plus loin

Pour faire un jeu complet de bataille navale ou le joueur affronte l'ordinateur :

* il faut deux mers ;
* la fonction suivante permet d'initialiser de façon aléatoire la position des navires de l'ordinateur ; pour la comprendre, en faire une trace avec dimensions = 3.

public function **initialiserNavires**(mer : Mer, dimensions : Int) { **//classe Main**

var choix : Array<Dynamic> = new Array();

for (i in 0...dimensions) { //on crée un tableau avec toutes les positions possibles

for (j in 0...dimensions) {

choix.push({ ligne : i, colonne : j });

}

}

for (i in 0...NBNAV) { //on crée le nombre de navires requis

var alea : Int = Std.random(choix.length); //on tire une position au hasard

var coord : Dynamic = choix[alea];

choix.remove(coord); //on supprime l'élément du tableau pour ne pas le retirer

var n : Navire = new Navire(coord.ligne, coord.colonne, mer);

mer.ajouterNavire(n);

}

}

*Les classes permettant d'afficher du texte ou de jouer des fichiers audio seront vues ultérieurement.*

Ce jeu pourra fonctionner sous Windows, Linux, Mac OS, sur smartphone (Android, IOS, …) ou dans le navigateur (Flash ou HTML5).