Programmation Orientée Objet - Partie II
L'art de la POO

Par Nicolas Hurtubise
Ajouts de Sébastien Roy

IFT1025 - Programmation 2

Au programme...

- Exercices pratiques
- Couplage et cohésion
- Principe d'encapsulation
- Packages
- Programmation orientée objet vs Programmation procédurale

Exercices

Exercices : essayez de nommer les objets dans les programmes suivants

Exercice : Super Mario

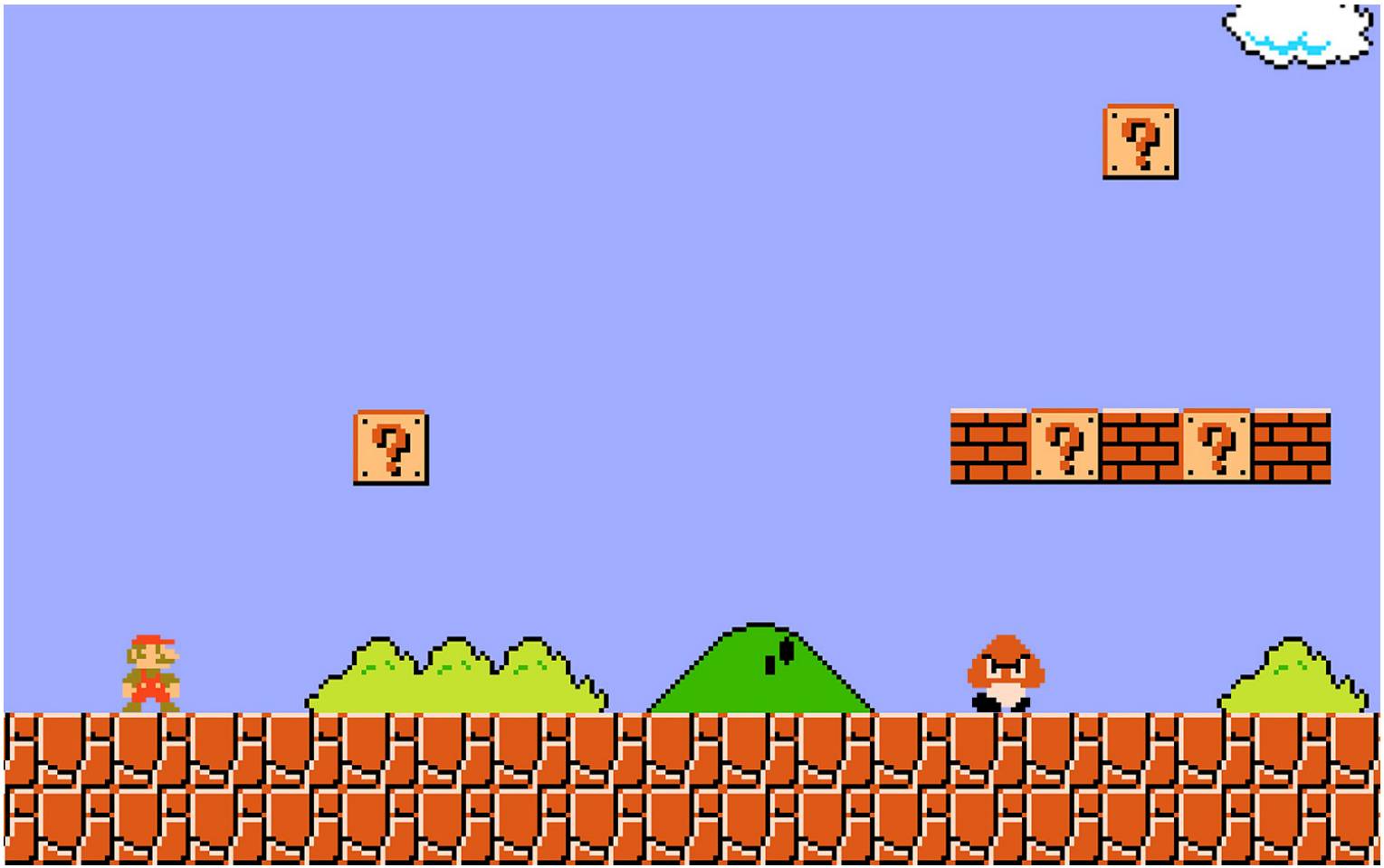


Figure 1:
https://www.nintendo.fr/Jeux/NES/Super-Mario-Bros--803853.html

Exercice : Super Mario

Idées...

```
class Mario
    - Etat (petit/grand/fleur)
    - Nombre de vies
    - Nombre de points

class Bloc
    - Qui contient un item ou non
    - Quel item est contenu (champignon/pièce)

class Goomba
    - Vitesse, direction

class Level
    - Numéro (ex.: level 1-01)
    - Temps restant pour le finir

class Champignon
class Piece
```

Exercice : Échecs

Figure 2:
https://commons.wikimedia.org/wiki/File:ChessStartingPosition.jpg

Exercice : Échecs

Idées...

```
class Partie
    - 2 joueurs
    - 1 échiquier

class Joueur
    - Pièces

class Echiquier
    - Liste des pièces

class Pion
    - Couleur (blanc/noir)
    - Position
class Fou
class Cavalier
class Tour
class Reine
class Roi
```

Exercice : Battle Ship

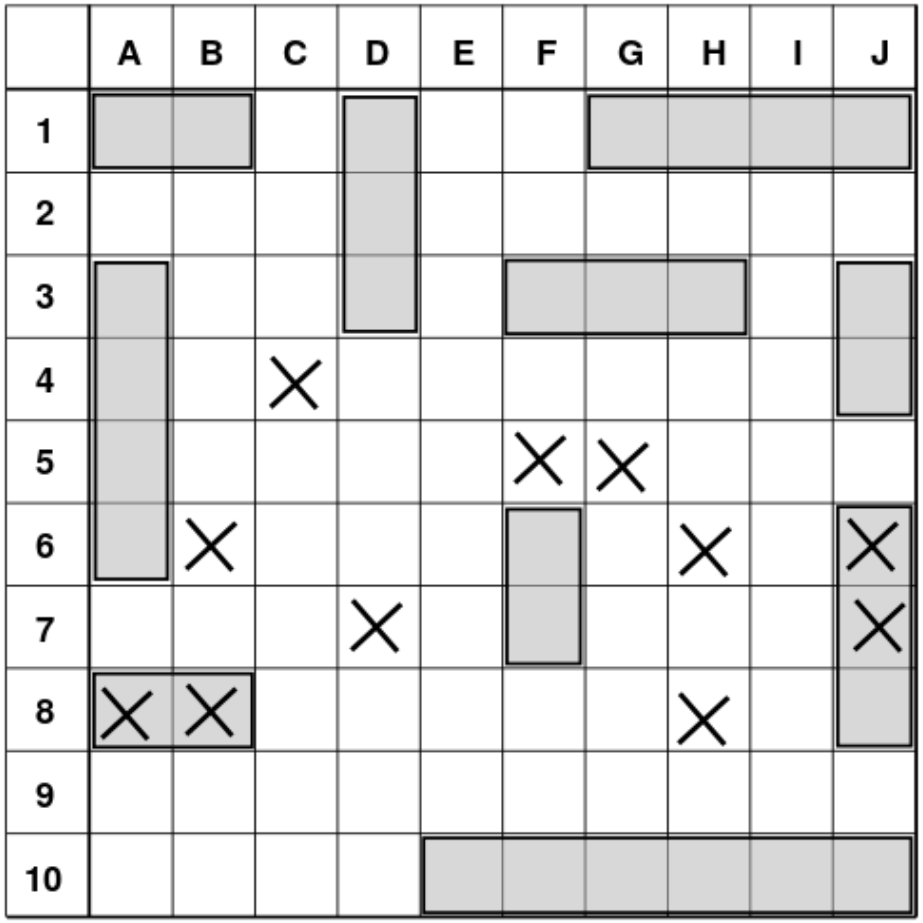


Figure 3: Par Actam — Travail personnel, GFDL, https://commons.wikimedia.org/w/index.php?curid=3581865

Exercice : Battle Ship

Idées...

```
class Jeu
    - 2 joueurs
    - Une grille

class Joueur
    - Joueur 1 ou 2
    - Liste de bateaux
    - Liste de missiles envoyés à l'adversaire

class Grille
    - Liste des bateaux du jeu

class Bateau
    - Position x/y sur la grille
    - Taille (2/3/4 cases)
    - Orientation (horizontal/vertical)
    - Missiles reçus
...
```

Couplage & Cohésion

Il y a visiblement plusieurs façons différentes de réfléchir en termes d'objets

Pour bien concevoir une ensemble de classes :

- Maximiser la cohésion
  - Chaque classe représente un seul concept
  - Toutes les composantes de la classe sont étroitement reliés à ce concept central
- Minimiser le couplage
  - Couplage entre des classes `X` et `Y` si `X` utilise un objet de classe `Y`
  - Classe `X` dépend de classe `Y` pour fonctionner

Couplage & Cohésion

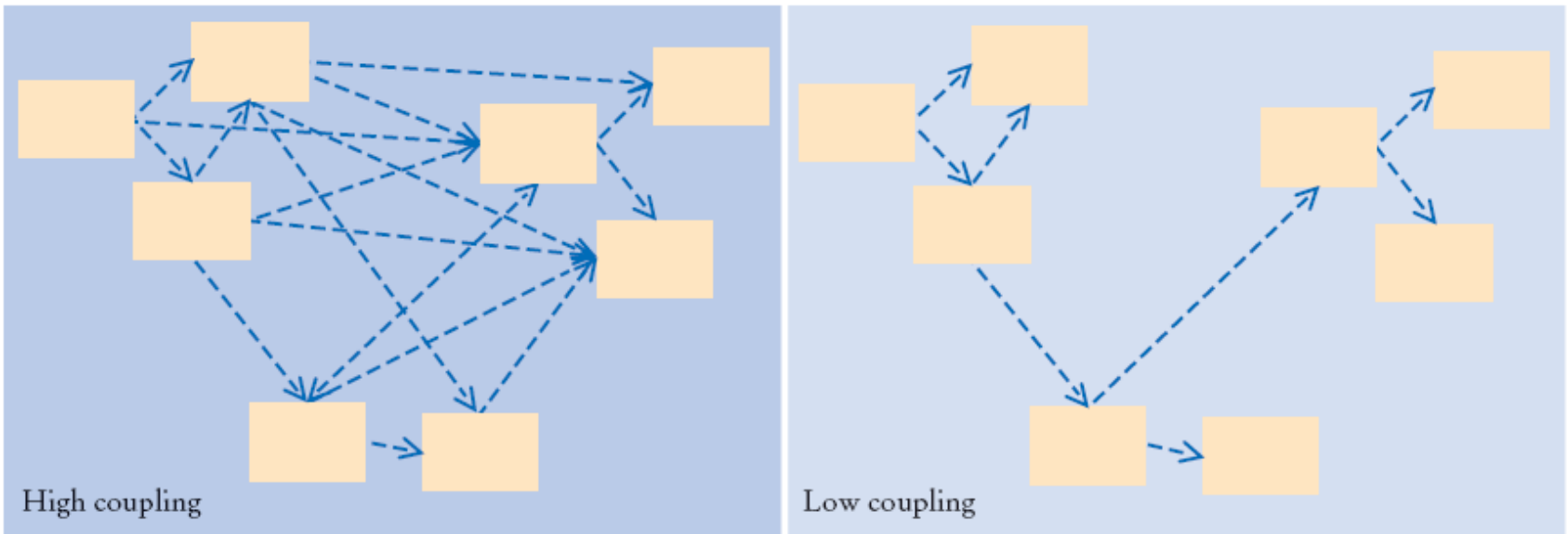


Figure 2 High and Low Coupling Between Classes

Figure 4: Figure tirée du livre Big Java

Couplage & Cohésion

Exemple : on veut envoyer une Newsletter universitaire aux étudiants et professeurs abonnés

```
public class Etudiant {
    public String prenom, nom;
    public int matricule;

    // ...
}

public class Professeur {
    public String prenom, nom;
    // ...
}
```

Couplage & Cohésion

```java
public class Newsletter {
    public String contenu;
    // Envoie la newsletter universitaire aux abonnés
    public void envoyer(Etudiant[] etudiants, Professeur[] profs) {

        // prenom.nom@umontreal.ca
        for(int i=0; i<etudiants.length; i++) {
          String courriel = etudiants[i].prenom + "." +
                            etudiants[i].nom + "@umontreal.ca";
          sendEmail(courriel, this.contenu);
        }

        for(int i=0; i<profs.length; i++) {
          String courriel = profs[i].prenom + "." +
                            profs[i].nom + "@umontreal.ca";
          sendEmail(courriel, this.contenu);
        }
    }
}
```

Couplage & **Cohésion**

Cohésion : toute tâche qui est reliée à une classe en particulier devrait être effectuée par celle-ci.

```
for(int i=0; i<etudiants.length; i++) {

    /* Calculer l'adresse email d'un étudiant
       devrait plutôt se faire dans la classe Etudiant */
    String courriel = etudiants[i].prenom + "." +
                      etudiants[i].nom + "@umontreal.ca";

    sendEmail(courriel, this.contenu);
}
```

Couplage & **Cohésion**

```java
public class Etudiant {
    public String prenom, nom;
    public int matricule;

    public String getCourriel() {
        return this.prenom + "." + this.nom + "@umontreal.ca";
    }
    // ...
}
```

Couplage & **Cohésion**

```java
public class Newsletter {
  public String contenu;

  // Envoie la newsletter universitaire aux abonnés
  public void envoyer(Etudiant[] etudiants, Professeur[] profs) {

    // prenom.nom@umontreal.ca
    for(int i=0; i<etudiants.length; i++) {
      String courriel = etudiants[i].getCourriel();
      sendEmail(courriel, this.contenu);
    }

    for(int i=0; i<profs.length; i++) {
      String courriel = profs[i].getCourriel();
      sendEmail(courriel, this.contenu);
    }
  }
}
```

Couplage & Cohésion

Couplage : il faut minimiser la dépendence entre les différentes classes

```
for(int i=0; i<etudiants.length; i++) {
    String courriel = etudiants[i].getCourriel();
    sendEmail(courriel, this.contenu);
}

for(int i=0; i<profs.length; i++) {
    String courriel = profs[i].getCourriel();
    sendEmail(courriel, this.contenu);
}
```

Est-ce que la dépendance entre `Newsletter` et `Professeur`/`Etudiant` est nécessaire ?

Couplage & Cohésion

```java
public class Newsletter {
    public String contenu;
    // Envoie la newsletter universitaire aux abonnés
    public void envoyer(String[] courriels) {
        for(int i=0; i<courriels.length; i++) {
            sendEmail(courriels[i], this.contenu);
        }
    }
}
```

- Cette classe ne dépend pas de l'existence des classes `Etudiant` et `Professeur`
- => beaucoup plus simple de l'intégrer dans des nouveaux projets
  - ex.: l'utiliser pour envoyer la Newsletter d'un organisme communautaire qui n'a aucune notion d'étudiants/professeurs

Exercice : Date

Modéliser une Date simple :

- Permettre d'afficher la date dans un format standard
- Permettre de calculer le jour de la semaine

```
Date jour = new Date(2018, 1, 17); // 17 janvier 2018

System.out.println(jour.dateStandard());
// => "2018-01-17"
System.out.println(jour.dateAmericaine());
// => "01/17/2018"

System.out.println(jour.getJourDeLaSemaine());
// => 3 (ou "mercredi")
```

(Voir le cours de Programmation 1 pour la formule du jour en fonction de l'année, du mois, du quantième...)

Exercice : Date

Permettre de calculer la différence entre deux dates :

```
Date jour1 = new Date(2018, 2, 1);
Date jour2 = new Date(2018, 3, 1);

System.out.println(Date.joursEntre(jour1, jour2));
// => 28

Date jour3 = new Date(2018, 3, 1);
Date jour4 = new Date(2018, 4, 1);

System.out.println(Date.joursEntre(jour3, jour4));
// => 31
```

Exercice : Date

Permettre d'ajouter un nombre de jours :

```
Date jour = new Date(2018, 1, 17); // 17 janvier 2018

int nbrJours = 3;

jour.ajouter(nbrJours);

System.out.println(jour.dateStandard());
// => "2018-01-20"
```

Exercice : Date

Supposons qu'on modélise la date de la façon suivante :

```
public class Date {
    public int quantieme;
    public int mois;
    public int annee;

    public void ajouterJours(int nbrJours) {
        // On ajuste le quantième, le mois et l'année
        // ...
    }


    // ...
}
```

Exercice : Date

Qu'est-ce qui se passe si on fait quelque chose comme ceci ?

```
Date date = new Date(2018, 1, 30); // 30 janvier 2018

// Ajouter 2 jours
date.quantieme += 2;

System.out.println(date.dateStandard());
// => 2018-01-32 ?
```

Notre programme peut tomber dans un *état incohérent* dû à une mauvaise utilisation de la classe...

Le Principe d'Encapsulation

- Solution ? Mettre des garde-fous !
- Toute modification des attributs de l'objet **doit** se faire via des méthodes de l'objet plutôt que directement
- On peut interdire l'accès à des attributs depuis l'extérieur de la classe en utilisant le mot-clé `private` plutôt que `public`

```
public class Date {
    private int quantieme;
    private int mois;
    private int annee;

    ...
}
```

- La seule façon d'ajouter des jours est alors de passer par la méthode `public ajouterJours`, qui va se charger de garder les attributs cohérents

Le Principe d'Encapsulation

Si on veut manipuler les attributs individuels, on définit :

- Des accesseurs (ou getters) : `getTruc()`, pour accéder aux attributs
- Des mutateurs (ou setters) : `setTruc()`, pour modifier les attributs

```
public class Etudiant {
    private int quantieme, mois, annee;

    public String getQuantieme() {
        return prenom;
    }

    public void setQuantieme(int q) {
        /* Valider que le quantième est entre 1 et 31 et que
           le nombre est cohérent avec le mois actuel
           (pas de 31 février !)

           ...
        */
    }
```

Le Principe d'Encapsulation

Généralement :

- Attributs `private`
- Méthodes `public`

- On peut avoir une méthode `private`, si on ne souhaite pas exposer un traitement aux utilisateurs de la classe
- On a rarement des attributs d'objets `public`, mais c'est possible dans certains cas
  - Ex.: une classe qui sert purement à **stocker des données structurées**, sans aucune méthode

Le Principe d'Encapsulation

Exemple : une classe décrivant un Point 2D quelconque

```
public class Point {
    public double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
}
```

Équivalent à un enregistrement en JavaScript : tous les attributs sont `public`

```
Point p = new Point(44, 64);

System.out.println(p.x); // => 44
p.y += 10;
```

Le Principe d'Encapsulation

Même si on n'a pas de restrictions particulières sur les valeurs que peuvent prendre x et y, on pourrait quand même décider d'appliquer le principe d'encapsulation :

```
public class Point {
    private double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }

    ...
```

Le Principe d'Encapsulation

```
    ...
    // Accesseurs
    public double getX() {
        return this.x;
    }

    public double getY() {
        return this.y;
    }

    // Mutateurs
    public void setX(double x) {
        this.x = x;
    }

    public void setY(double y) {
        this.y = y;
    }
}
```

Le Principe d'Encapsulation

Pourquoi ne pas définir les attributs `public` si on ne fait pas de validation et qu'un accès direct aux attributs ne poserait pas de problème ?

Le Principe d'Encapsulation

Pourquoi ne pas définir les attributs `public` si on ne fait pas de validation et qu'un accès direct aux attributs ne poserait pas de problème ?

- Si jamais on a besoin de changer l'implantation un jour, l'interface publique reste la même

```
Point :
  - getX(), setX(x)
  - getY(), setY(y)
```

Le Principe d'Encapsulation

Pourquoi ne pas définir les attributs `public` si on ne fait pas de validation et qu'un accès direct aux attributs ne poserait pas de problème ?

- Si jamais on a besoin de changer l'implantation un jour, l'interface publique reste la même

```
Point :
  - getX(), setX(x)
  - getY(), setY(y)
```

- Rien ne nous empêche de modifier les rouages internes de notre objet dans la version 2 de notre classe, tant que l'objet s'utilise de la même façon
  - Ex.: Vérifier que X et Y sont positifs

Le Principe d'Encapsulation

```
public class Point {

    private double r, phi; // Coordonnées polaires

    public Point(double x, double y) {
        this.r = Math.sqrt(x*x + y*y);
        this.phi = Math.atan2(y, x);
    }

    public double getX() {
        return this.r * Math.cos(this.phi);
    }

    public double getY() {
        return this.r * Math.sin(this.phi);
    }
    ...
}
```

Le Principe d'Encapsulation

- En bref : un objet est une *Boîte noire*
- On sait comment l'utiliser via son interface publique
  - Les méthodes déclarées `public`
- On ne sait pas comment l'état est géré à l'interne
  - => On ne veut pas y toucher nous-même, pour éviter de le briser
  - Les attributs sont `private`

- Objectif : prévenir les bugs. On s'assure que l'objet ne se retrouvera pas dans un état incohérent à cause d'une mauvaise utilisation de la part des programmeurs qui utilisent l'objet

Le Principe d'Encapsulation

Question

```
public class Question {
    public int[] valeurs;
}
```

- Est-ce que ça respecte le principe d'encapsulation ?

Le Principe d'Encapsulation

Question

```
public class Question {
    private int[] valeurs;

    public int[] getValeurs() {
        return this.valeurs;
    }

    public void setValeurs(int[] valeurs) {
        this.valeurs = valeurs;
    }
}
```

- Est-ce que ça respecte le principe d'encapsulation ?

Le Principe d'Encapsulation

Question

```java
public class Question {
    private int[] valeurs;

    public int[] getValeurs() {
        // Créer un nouveau tableau en copiant this.valeurs
        int[] v = this.valeurs.clone();
        return v;
    }

    public void setValeurs(int[] valeurs) {
        this.valeurs = valeurs.clone();
    }
}
```

- Est-ce que ça respecte le principe d'encapsulation ?

Organisation des classes

Un gros projet aura typiquement *beaucoup* de classes

```
// Classes tirées du jeu RPG Pixel Dungeon :
Assets                CellEmitter      Speck          Item
Badges                CheckedCell      SpellSprite    KindOfWeapon
Bones                 DeathRay         Splash         LloydsBeacon
Challenges            Degradation      Swap           TomeOfMastery
Chrome                Effects          TorchHalo      Torch
Dungeon               EmoIcon          Wound          Weightstone
DungeonTilemap        Enchanting       Amulet         Dreamweed
FogOfWar              Fireball         Ankh           Earthroot
GamesInProgress       Flare            ArmorKit       Fadeleaf
Journal               FloatingText     Bomb           Firebloom
PixelDungeon          Halo             Dewdrop        Icecap
Preferences           IceBlock         DewVial        Plant
Rankings              Identification   EquipableItem  Rotberry
ResultDescriptions    Lightning        Generator      Sorrowmoss
Statistics            MagicMissile     Gold           Sungrass
BadgeBanner           Pushing          Heap
BannerSprites         Ripple           Honeypot ... et 439 autres!
```

Organisation des classes

- Comment s'y retrouver dans autant de fichiers ?

- Quoi faire si on a deux classes qui devraient intuitivement porter le même nom ?
  - `Fire` pour la classe d'une boule de feu (attaque magique qu'on envoie sur un ennemi) *et* pour du feu qui brûle sur le sol (élément de décor)
  - `Poison` pour la classe d'une potion de poison *et* pour la classe d'un enchantement d'empoisonnement
  - ...

Organisation des classes

Java utilise la notion de `packages` et de sous-`packages`

- `package` == regroupement de plusieurs classes qui ont un sens commun
  - Le jeu *Pixel Dungeon* aura son propre package
  - Les classes qui définissent les items du jeu partageront un sous-`package`
  - Les classes de personnages seront définis dans un sous-`package` différent
- Un programme peut être composé d'autant de `packages` que nécessaire
- Chaque librairie externe que le projet utilise aura également son propre `package`

Organisation des classes

Utilisation : on doit déclarer le `package` auquel un fichier appartient au tout début, avant la déclaration de classe

Boule de feu magique

```java
// Fichier com/watabou/pixeldungeon/items/Fire.java
package com.watabou.pixeldungeon.items;

// Item magique : boule de feu
public class Fire {
    // Attributs et méthodes
    // ...
}
```

Le nom de `package` **doit** suivre la hiérarchie de fichiers (dossiers/sous-dossiers dans le code source du projet)

Organisation des classes

Utilisation : on doit déclarer le `package` auquel un fichier appartient au tout début, avant la déclaration de classe

Feu de bois dans le décor du jeu

```java
// Fichier com/watabou/pixeldungeon/backgrounds/Fire.java
package com.watabou.pixeldungeon.backgrounds;

// Élément de décor : feu de bois
public class Fire {
    // Attributs et méthodes
    // ...
}
```

Le nom de `package` **doit** suivre la hiérarchie de fichiers (dossiers/sous-dossiers dans le code source du projet)

Organisation des classes

Convention : on nomme généralement un `package` en suivant le nom de domaine de l'entreprise ou de l'organisation qui s'occupe de gérer le code, *à l'envers*

Si le site web de l'équipe de *Pixel Dungeon* est watabou.com, on aurait le package principal

```
package com.watabou.pixeldungeon;
//      nom de domaine (à l'envers) . nom du projet
```

Et les sous-packages

```
package com.watabou.pixeldungeon.sous.package;
//      nom de domaine (à l'envers) . noms des sous-packages
```

Organisation des classes

Autre exemple, le programme `Hello World` développé par l'utilisateur `hurtubin` au *DIRO* à l'UdeM :

```java
// Projet : Hello World
// Fichier : ca/umontreal/iro/hurtubin/helloworld/Hello.java
package ca.umontreal.iro.hurtubin.helloworld;

public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Organisation des classes

Pour compiler et exécuter une des classes :

```
# Depuis le dossier source du projet
javac com/watabou/pixeldungeon/backgrounds/Fire.java
java com.watabou.pixeldungeon.backgrounds.Fire
```

Notez : tout ça est généralement automatisé dans les IDEs (NetBeans/Eclipse/IntelliJ/...)

Organisation des classes

De base, seules les classes du package actuel sont accessibles

Pour utiliser une classe d'un autre package, on peut soit :

- Utiliser son nom complet

```
package ca.umontreal.iro.hurtubin.testscanner;




public class Test {
    public static void main(String[] args) {
        java.util.Scanner scanner =
            new java.util.Scanner(System.in);

        // ... Utiliser le scanner
```

Organisation des classes

De base, seules les classes du package actuel sont accessibles

Pour utiliser une classe d'un autre package, on peut soit :

- L'importer dans le fichier pour y avoir accès comme si elle faisait partie du package actuel

```
package ca.umontreal.iro.hurtubin.testscanner;

/* Importe la classe Scanner qui se trouve
   dans le package java.util */
import java.util.Scanner;

public class Test {
    public static void main(String[] args) {
        // Un peu plus clair
        Scanner scanner = Scanner(System.in);

        // ... Utiliser le scanner
```

Organisation des classes

Encore une fois, ceci est automatisé dans la plupart des IDEs : taper simplement `Scanner` (ou toute autre classe de la librairie de Java ou du projet) proposera d'importer la classe pour l'utiliser.

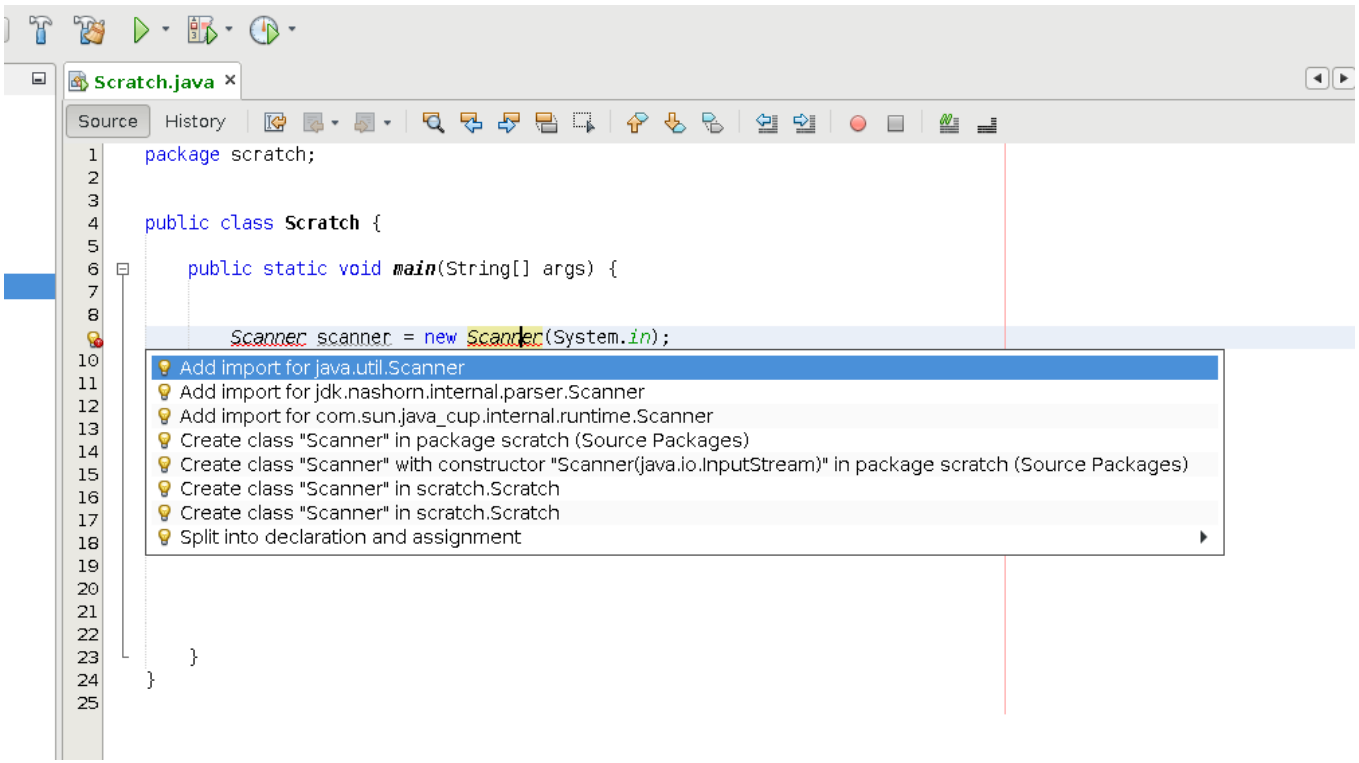


Figure 5: Auto-complétion de l'importation classe dans NetBeans (Alt+Enter sur le nom de la classe)

Orienté Objet vs Procédural

- Est-ce que l'Orienté Objet est *objectivement et purement meilleur* que le Procédural ?

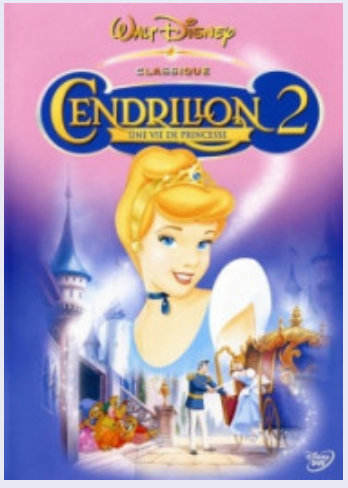
Oui ?

- On voit ça en Programmation 2, donc `Prog 2 > Prog 1`, donc c'est sûrement mieux ?
- Ça a été inventé plus tard, donc forcément, c'est une amélioration sur ce qui existait avant ?

Orienté Objet vs Procédural

Non !

- Cendrillon 2 ?
- Star Wars Prequels inventées après la trilogie originale...



Principe : simplicité

```java
// Fichier : NeRienFaire.java
public class NeRienFaire {
    public static void main(String[] args) {
        // Programme qui ne fait rien
    }
}
```

- Est-ce que ce programme contient un bug ?

Principe : simplicité

```java
// Fichier : NeRienFaire.java
public class NeRienFaire {
    public static void main(String[] args) {
        // Programme qui ne fait rien
    }
}
```

- Est-ce que ce programme contient un bug ?

Clairement pas !

Principe : simplicité

```java
// Fichier: Salutation.java
public class Salutation {
    private String formulation;

    public Salutation(String formulation) {
        this.formulation = formulation + ", ";
    }

    public void setSalut(String formulation) {
        this.formulation = formulation + ", ";
    }

    public String getSalut() {
        return this.formulation;
    }
    public String saluer(String personne) {
        return this.formulation + personne + " !";
    }
}
```

Principe : simplicité

(suite)

```java
// Fichier: Hello.java
public class Hello {
    public static void main(String[] args) {
        Salutation salut = new Salutation("Hello");
        salut.saluer("World"); // Afficher "Hello, World !"
    }
}
```

- Est-ce que ce programme contient un bug ?

Principe : simplicité

(suite)

```java
// Fichier: Hello.java
public class Hello {
    public static void main(String[] args) {
        Salutation salut = new Salutation("Hello");
        salut.saluer("World"); // Afficher "Hello, World !"
    }
}
```

- Est-ce que ce programme contient un bug ?

Peut-être ? Il faudrait tester...

Principe : simplicité

L'exemple précédent est adapté de la présentation intitulée *Stop Writing Classes*

> *I hate code and I want as little of it as possible in our product*
>
> ~ Jack Diederich, *Stop Writing Classes*

> *Simplicity is prerequisite for reliability*
>
> ~ Edsger W. Dijkstra

Principe : simplicité

- Généralement : plus on a de code, plus on a de bugs
- Moins de code =>
  - Plus rapide à lire et à comprendre
  - Plus facile de se convaincre que le code est bon en le lisant
  - Moins de bugs en général

Principe : simplicité

- *Attention* : le but n'est pas seulement d'avoir moins de lignes de code au total
- Du code illisible est pire que trop de code pour tout lire !

The competent programmer is fully aware of the strictly limited size of his own skull

~ Edsger W. Dijkstra

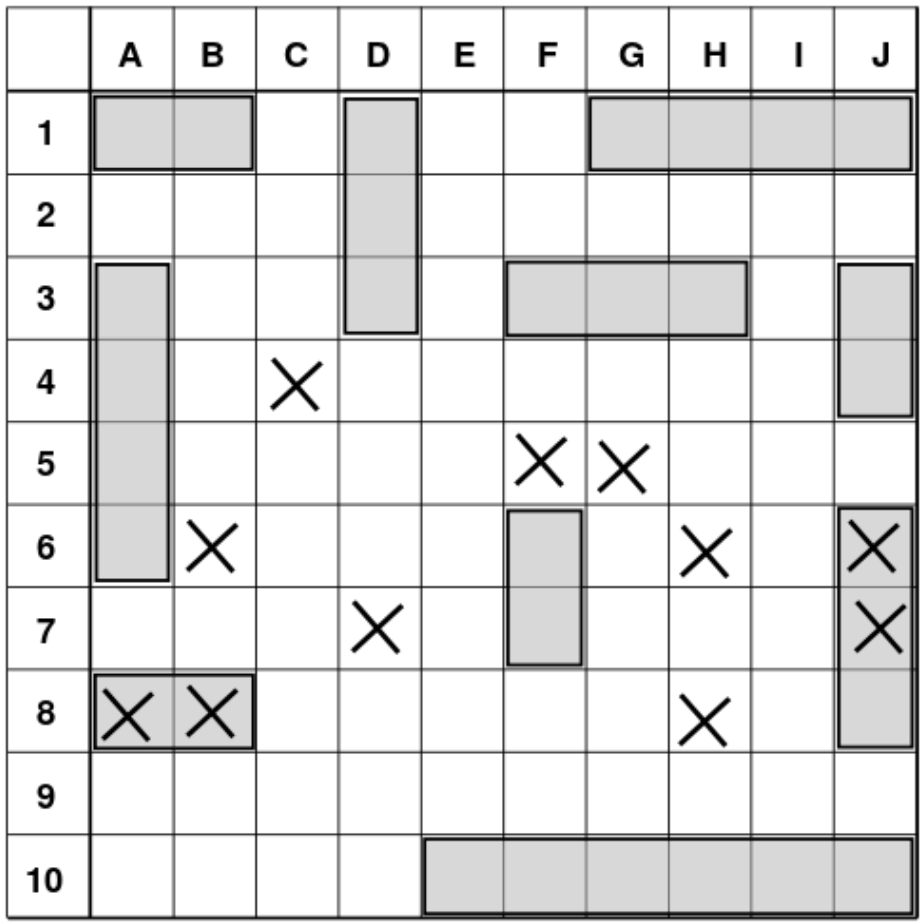
```java
// Afficher les 10 premières puissances de 2
for(int i=-1,j=1;i<10;System.out.println(++i+":"+((j <<= 1)>>1)));
```

Principe : simplicité

- La Programmation Orientée Objet est un outil puissant, mais complexe à gérer
- Il faut savoir choisir les bons outils !
  - On ne construit pas un château de sable avec un bulldozer
  - On ne prépare pas le terrain pour la construction d'un édifice avec une pelle

Principe : simplicité

Revenons sur l'exemple de Battle Ship...



Principe : simplicité

```
class Jeu
    - 2 joueurs
    - Une grille

class Joueur
    - Joueur 1 ou 2
    - Liste de bateaux
    - Liste de missiles envoyés à l'adversaire

class Grille
    - Liste des bateaux du jeu

class Bateau
    - Position x/y sur la grille
    - Taille (2/3/4 cases)
    - Orientation (horizontal/vertical)
    - Missiles reçus
...
```

Principe : simplicité

- Est-ce qu'un bateau devrait connaître sa position, ou est-ce que la grille pourrait déjà contenir cette information ?
- As-t-on besoin de distinguer les différents bateaux une fois la partie commencée ?
- Est-ce que ces objets aident à mieux structurer le code ?

Principe : simplicité

On pourrait remplacer tout ça par seulement trois variables :

```
int tour = 1; // 1 ou 2, indique qui doit jouer

int[][] grille1 = ...; // Grille du joueur 1
/**
    0: case vide
    1: bateau du joueur 1
    2: bateau touché

    0 1 0 0 0 0
    0 1 0 0 0 0
    0 1 0 1 2 2
    1 0 0 0 0 0
    1 0 1 1 2 2
*/
int[][] grille2 = ...; // Grille du joueur 2
```

Principe : simplicité

- Doit-on utiliser de l'orienté objet pour résoudre un problème donné ?

- Oui, *si et seulement si* ça simplifie l'implémentation et/ou la maintenance du code
- Parfois, la programmation procédurale est très bien adaptée à résoudre le problème
- D'autres fois, l'orienté objet simplifie beaucoup la conception de la solution
- Il faut s'adapter et utiliser la solution la plus simple qui répond aux besoins

Principe : simplicité

- Comment déterminer **la** solution la plus simple ?
  - Pas de solution miracle
  - Prendre le temps d'y réfléchir avant de se mettre à programmer
  - Programmer beaucoup, acquérir de l'expérience pour mieux juger les prochaines fois

(Parenthèse au sujet de Dijkstra - Pas à l'examen)

- Très influent dans le monde de l'informatique, pionnier sur plusieurs sujets en informatique, autant théorique qu'en génie logiciel
- Très amusant à citer
- Très intéressant à lire
- À prendre au second degré...

> *It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.*

Edsger W. Dijkstra

(Parenthèse au sujet de Dijkstra - Pas à l'examen)

> *You probably know that arrogance, in computer science, is measured in nanodijkstras.*

Alan Kay, OOPSLA 1997

- Plus de citations par/sur Dijkstra ici : https://en.wikiquote.org/wiki/Edsger_W._Dijkstra
- Notes personnelles de Dijkstra : https://www.cs.utexas.edu/ EWD/transcriptions/transcriptions.html

Pour conclure

Est-ce que l'Orienté Objet == Écrire et utiliser des classes ?