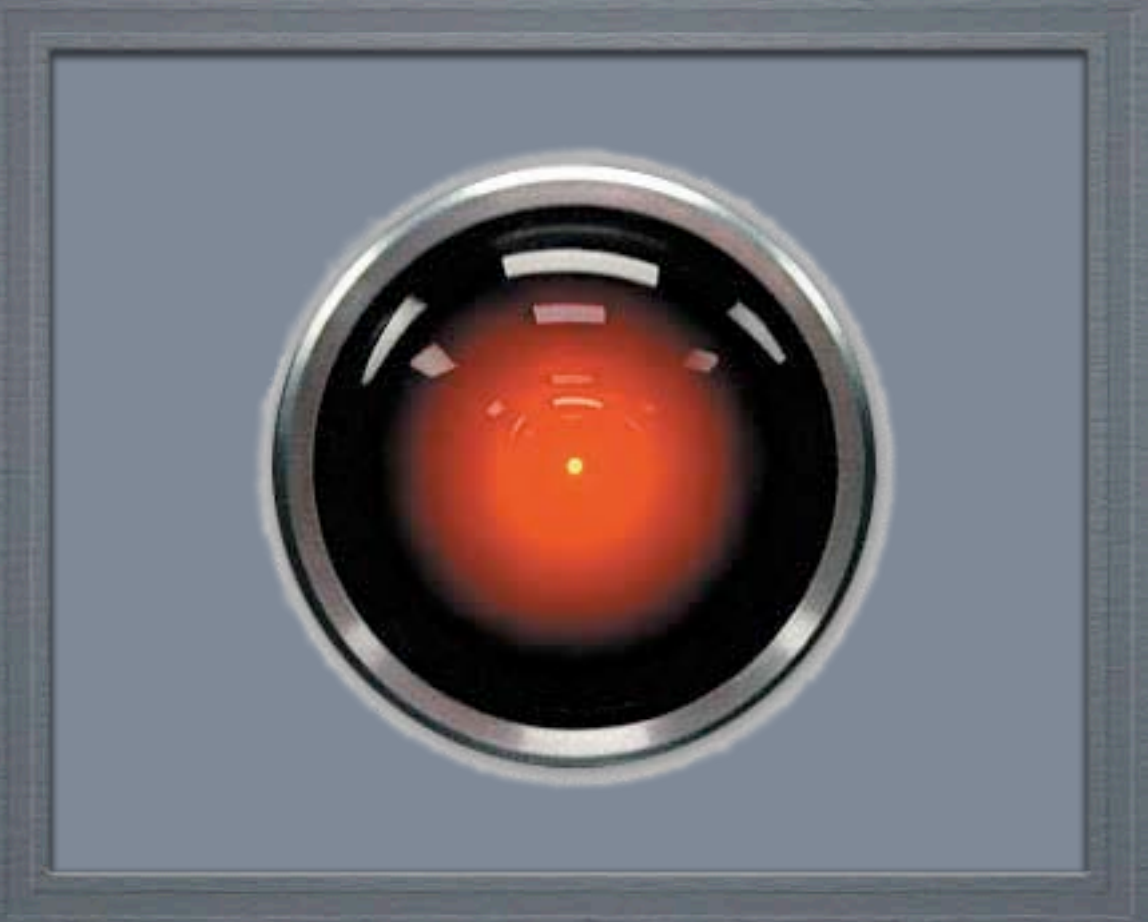


# Hello ~~Dave~~

Nicolas

# Organisation

- Équipe enseignante
- Planning
- Projet

# Programmation Orientée Objet

## Let's get started

# Vers la programmation objet

# Rappels

- Une fonction est une succession d'instructions basiques effectuées dans l'ordre et toujours de la même façon
- Une fonction informatique peut être représentée par une fonction mathématique, ou une succession de boites noires.
- Une fonction simple s'exprime sous la forme

  résultat = fonction(paramètre 1, paramètre 2, etc...)

  Une fonction a donc un domaine d'application, et un domaine de résultats

# Rappels II

- Une fonction plus compliquée peut modifier ses paramètres a l'aide des pointeurs (passage par adresse)
- Un programme est une succession de fonctions
- Si en plus on ajoute la possibilité d'exécuter des fonctions en parallèle (threads), on se retrouve avec un modèle complexe

# Vers un nouveau paradigme

- Pour pallier a la complexité de la programmation séquentielle il a fallu chercher un autre moyen de représenter un programme
- La programmation séquentielle se base sur le principe de l'Ordonnateur, une espèce de super entité dans un programme qui contrôlerait tout, et doit tout gérer
- La classification des fonctions en groupes (ou modules) se fait assez naturellement
- En se basant sur la modularité et en empruntant une gestion déléguée au monde physique qui nous entoure, on arrive a la programmation objet

# Objets

- Lorsqu'on pense au monde physique, on pense en objets
- Un objet a des propriétés de famille, et des propriétés intrinsèques

  Par exemple: toutes les planètes sont a peu près rondes, et la Terre fait environ 6500 km de rayon

# Objets II

- Une famille d'objets est appelée une classe d'objets

  Elle regroupe toutes les informations qui sont liées a tous les objets d'un même type

- Un objet en particulier est appelé une instance

  Il a des propriétés qui lui sont propres, et qui ne sont pas partages par les autres objets de la même classe

# Objets III

- La classe planète a donc une propriété "rayon" (puisque toutes les planètes sont des sphères), et la valeur de ce rayon varie en fonction de la planète concernée.
- Ces variables sont appelées variables d'instance
- Une variable de classe est une propriété dont la valeur ne change jamais, quelle que soit l'instance concernée.

  Par exemple, un tricycle a trois roues.

# Hiérarchie

- On peut identifier des sous groupes dans une famille.

  Par exemple, un 4x4 reste une voiture.

- On doit hiérarchiser les objets, de façon a identifier les propriétés communes et les spécificités de chacun

# Hiérarchie II

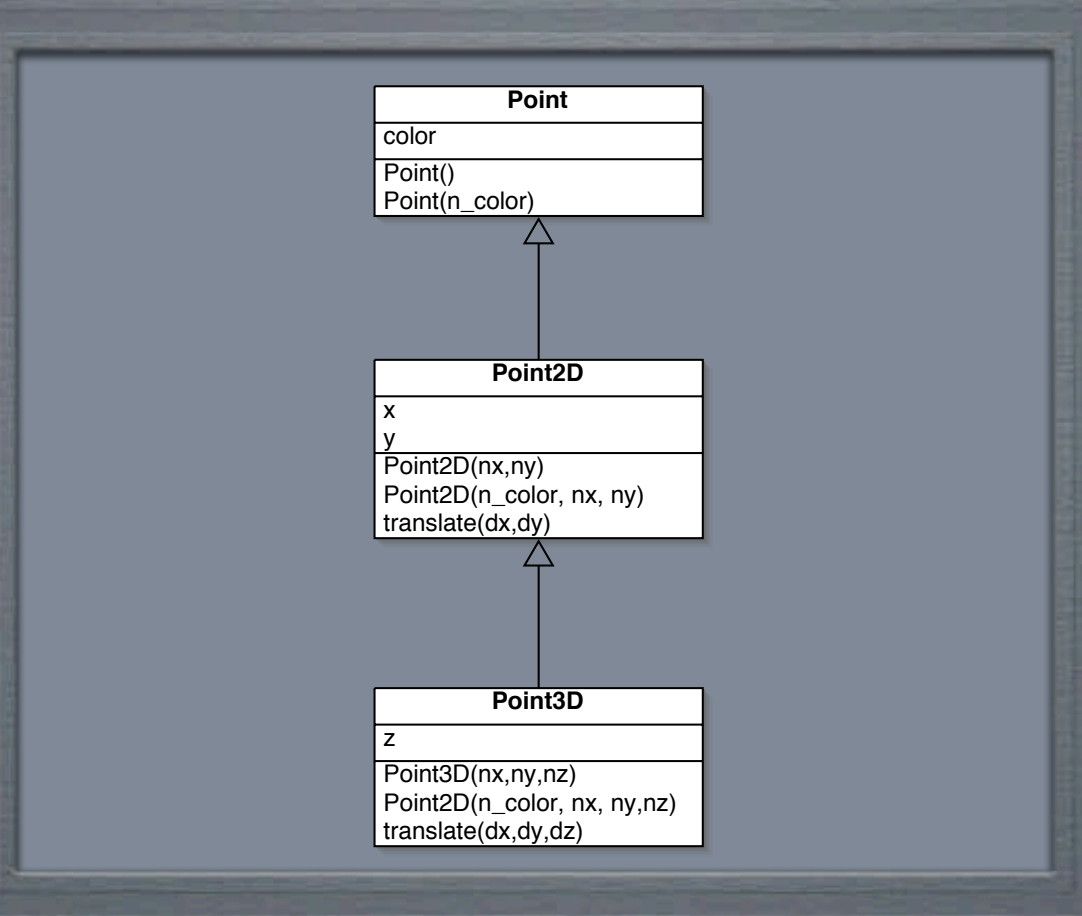
- Dans le cas des planètes, on en a deux grandes sortes: telluriques et gazeuses
- On peut donc avoir une classe planète gazeuse et une classe planète tellurique
- Faire une classe par objet existant *in potentia*?

# Hiérarchie III

- Classification en familles comme les animaux et les végétaux
- Une planète a donc un rayon.
  - Une planète tellurique est donc une planète avec une épaisseur de croûte (mais elle au aussi un rayon)
  - Une planète gazeuse est une planète avec une épaisseur d'atmosphère (et également un rayon)
- On appelle ça l'héritage : la classe planète est la classe mère de la classe planète tellurique, et la classe planète gazeuse est une classe fille de la classe planète

# Parenthèse

- Plutôt que de décrire systématiquement les classes, on a mis en place une façon de les dessiner
- UML (Unified Modeling Language)




# Hiérarchie de classes

# Agrégation

- Un objet peut en contenir d'autres (*faire référence à*)
- Exemple : le système solaire comprend 9 planètes (5 telluriques et 4 gazeuses)
- On appelle ce phénomène "agrégation"
- Une agrégation est une classe comme un autre, donc on peut aussi dériver des classes filles.

# Exemple géométrique

- Point
- Segment
- Polygone
- PointNommé
- SegmentNommé
- PolygoneNommé

# Intéractions

- Les objets dans notre univers interagissent entre eux
- Si on était en programmation séquentielle, notre Ordonnateur appliquerait des fonctions a nos objets
- Dans le monde physique, les interactions sont **locales**

# Intéractions II

- Dans le monde de la programmation objet, les intéractions (ou fonctions) sont appelées méthodes
- Exemple : déplacer un objet

  Un objet a une position intrinsèque (variable d'instance)

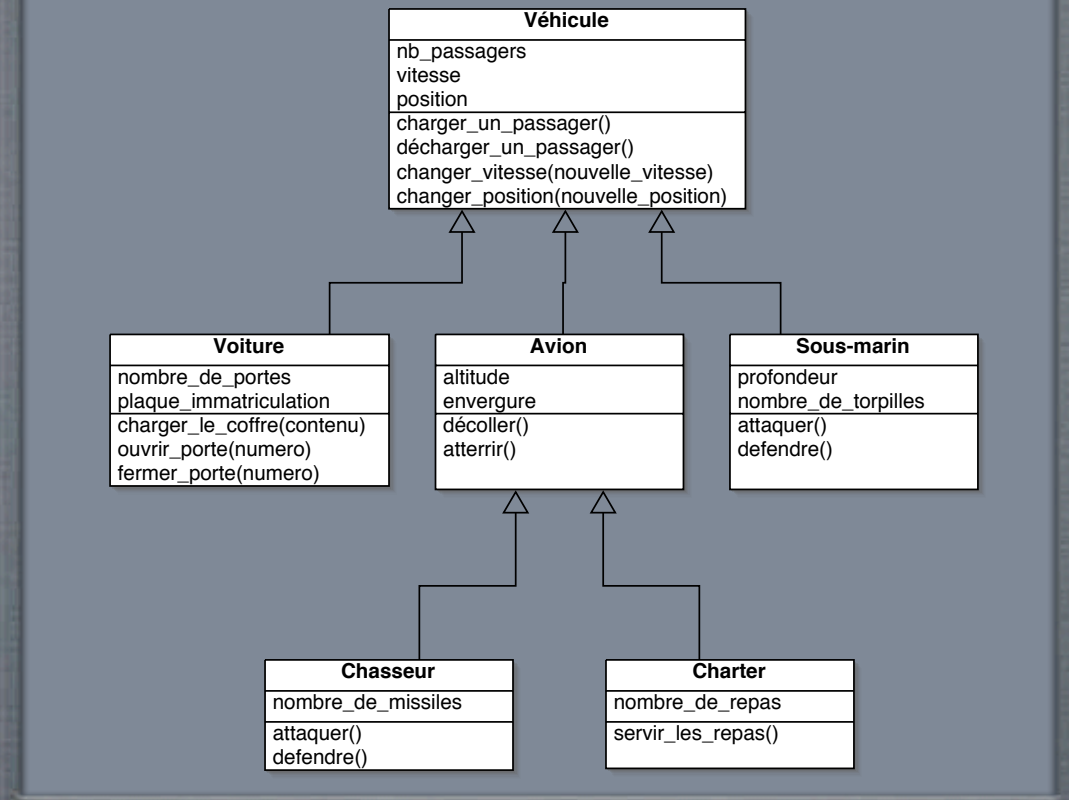
  Déplacer cet objet consiste à lui demander de bouger

# Intéractions III

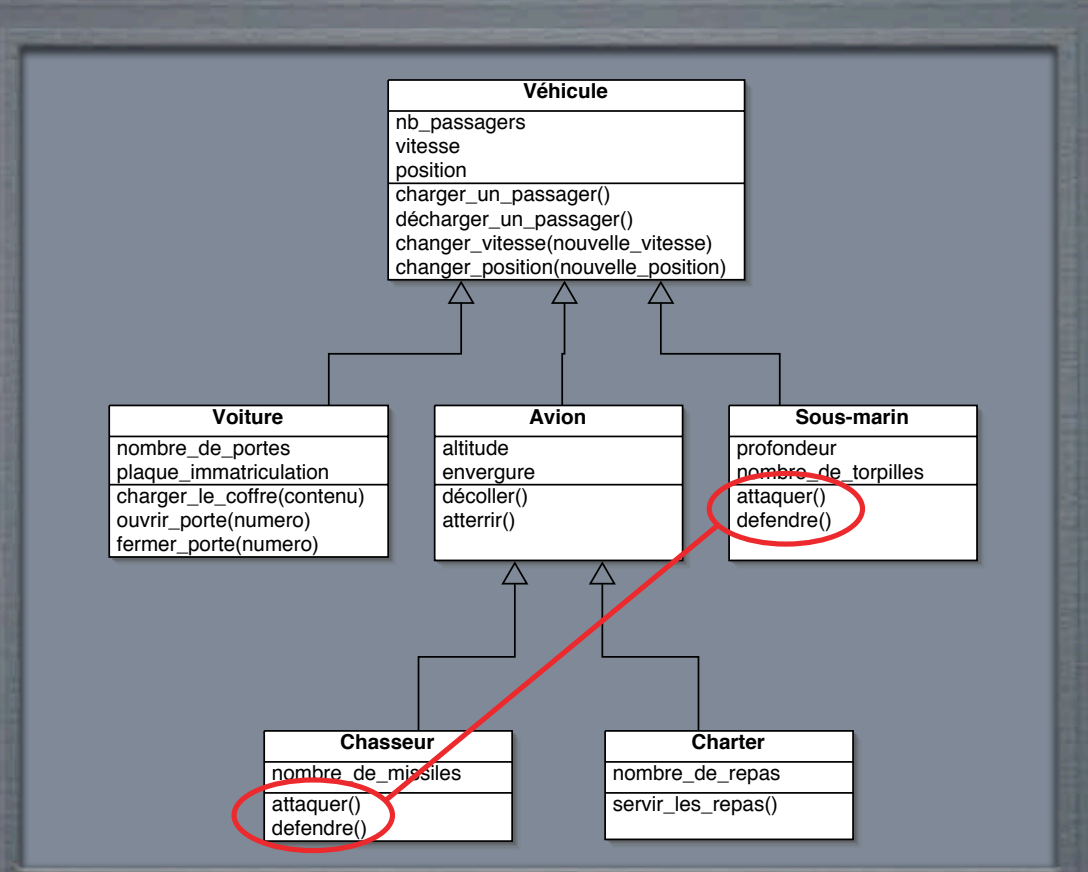
- Un objet peut (et doit) appeler les méthodes d'un autre objet pour le modifier
- Exemple : la gravitation : un objet déplace un autre objet en fonction de sa masse
- Notre Ordonnateur serait une espèce de classe agrégation contenant tous les objets de l'univers

# Hiérarchie d'intéractions

- Puisqu'on peut hériter des caractéristiques d'une classe d'objets, on peut aussi hériter de leur comportement
- Exemple : un véhicule peut charger un passager (donc rajouter un objet passager dans sa liste)
- Par conséquent, un bus, qui est aussi un véhicule peut également charger un passager, ainsi qu'une moto, un avion, ou un bateau


Véhicule
nb_passagers
vitesse
position
charger_un_passager()
décharger_un_passager()
changer_vitesse(nouvelle_vitesse)
changer_position(nouvelle_position)
Voiture
nombre_de_portes
plaque_immatriculation
charger_le_coffre(contenu)
ouvrir_porte(numero)
fermer_porte(numero)
Avion
altitude
envergure
décoller()
atterrir()
Sous-marin
profondeur
nombre_de_torpilles
attaquer()
defendre()
Chasseur
nombre_de_missiles
attaquer()
defendre()
Charter
nombre_de_repas
servir_les_repas()

# Héritage


Véhicule
nb_passagers
vitesse
position
charger_un_passager()
décharger_un_passager()
changer_vitesse(nouvelle_vitesse)
changer_position(nouvelle_position)
Voiture
nombre_de_portes
plaque_immatriculation
charger_le_coffre(contenu)
ouvrir_porte(numero)
fermer_porte(numero)
Avion
altitude
envergure
décoller()
atterrir()
Sous-marin
profondeur
nombre_de_torpilles
attaquer()
defendre()
Chasseur
nombre_de_missiles
attaquer()
defendre()
Charter
nombre_de_repas
servir_les_repas()

# Partage

# Spécifications

- Tout véhicule peut être armé, mais ne l'est pas forcément.
- On voudrait pouvoir définir une catégorie "militaire"

# Catégorie - Interface - Classe Abstraite

- On peut “forcer” un comportement à l’aide d’une spécification “externe”
- S’y conformer est obligatoire pour accéder à un nouveau statut
- On parle d’interface ou de classe abstraite quand cette définition ne force pas le “comment”, et de catégorie si le “comment” existe déjà.

# Interfaces

- En Java, on peut définir des classes à trous: une classe “presque définie”, pour laquelle un tout petit effort est réclamé pour qu’elle existe.
- Une classe complètement exprimée mais non définie est une interface.

# Petite parenthèse: UML

- L'implémentation d'une interface est souvent représentée par une flèche d'héritage en pointillés (non standard)

# Classes Abstraites

# Les interfaces "Utilitaires"

- Peut se rapporter à un comportement utilisable: sérialisation, lié aux processus concurrents, ...
- Peut spécifier une obligation de conformité (gestion de souris, de clavier, ...)

# Les interfaces "d'identification"

- Ne servent généralement qu'à mettre dans un même sac des objets disparates

# Différences avec d'autres langages

# Rappels

- Compiler un programme revient à "traduire" un langage dans un autre (généralement d'un langage "humainement compréhensible" dans un langage machine)
- Tous les langages "Turing complets" (qui peuvent exprimer toutes les fonctions mathématiques, en gros) sont inter-compilables.
- On peut dire qu'ils sont tous traduisibles et tous compilables, mais certains ne sont pas forcément traduits en langage machine

# Panorama

- Langages “compilables”:
  C++ / Java / Objective C /...
- Langages “à bytecode” ou “interprétés”:
  Java / C# / perl / python / ruby / PHP /...
- Curiosités:
  ocaml / NSOF / ...

# Lignes Générales

- Gestion mémoire (garbage collector, explicite, ou mixte)
- Indirections (structures, interprétation dynamique par un runtime,...)
- Typage (”Always trust the programmer”, typage “strict”)

# C++

- Syntaxe très proche de celle du C
- Superset du C
- Compilé avec des structures
- Gestion mémoire explicite (new / delete)

# C++ (II)

- Typage extrêmement relaxé, comme en C
- Passage des arguments en copie, pointeur ou valeur
- Héritage multiple et templates

# Objective C

- Superset du C avec un look & feel “smalltalk”
- Passage de “messages” aux objets
- Langage complètement dynamique
- Interfaces / Classes / Protocoles / Catégories

# Objective C (II)

- Gestion explicite de la mémoire (en théorie)
- Classe ancêtre commune
- Héritage mono-parental

# Pour le fun...

- Objective C++, un superset du C++ avec une méthodologie ObjC

# Langages "fonctionnels"

- Un exemple: CaML
- Tout élément de l'univers CaML est manipulable, y compris les fonctions
- Il existe une couche objet par dessus : OCaML
- Le fonctionnement des objets s'approche plus de l'ObjC que du C++

# Langages par prototype

- Toute petite classe de langages objets
- Instance = Classe
- Très faible empreinte mémoire, mais constante

# Java

- Gestion implicite de la mémoire (new existe mais pas delete)
- Typage assez strict
- Syntaxe assez proche du C
- WORA => intégration de la plupart des fonctionnalités au langage

# Conclusion(s)

- La programmation objet doit **FORCÉMENT** faire l'objet d'une reflexion avant écriture du code

- Mieux vaut avoir des objets génériques et des adaptations à des situations particulières que de gros objets polyvalents

- Si vous n'arrivez pas à traduire votre programme en français (ou en anglais), vous l'avez mal conçu

# Remarques diverses

- Le style de programmation vous est personnel. Toutefois certaines astuces rendent le code plus lisible et plus facilement maintenable:
  - Une indentation correcte vous permettra de vous relire
  - Un seul “return” par méthode. De cette façon, pas d’effet indésirable
  - Si vous écrivez plusieurs fois la même chose, le code peut être factorisé

# Java si vous avez fait du C

(aka "intro")

# Syntaxe

# Eléments du langage

- On utilise le point ( . ) pour appeler une méthode.
  Exemple : monObjet.maMéthode();
- “this” fait référence à l’instance courante et peut parfois être oubliée
- “super” fait référence à la classe mère

# Parenthèse : Scope

- On distingue 4 (3 en pratique) niveaux de visibilité
- public : accès non restreint pour tout le monde
- protected / package : accès réservé aux descendants ou aux membres du package
- private : accès réservé aux méthodes de la classe seulement

# Scope

- La portée ou le scope est utilisable devant chaque déclaration (variables, méthodes, classes, ... )

# Déclarations : variables

- <type> <nom de variable>;
- <type> <nom de variable> = <valeur initiale>;
- <type> <nom de variable> [ = ... ] , <nom de deuxième variable> , ... ;

# Déclaration : variables

- int i;
- char c = ‘a’;
- float x = 0.0, y = 1.0, z = 2.0;

# Déclarations : méthodes

- <type de retour> <nom de fonction>(<type d'argument 1> <nom d'argument 1>, ...)
  [ throws <type d'exception>];

- <type de retour> <nom de fonction>(<type d'argument 1> <nom d'argument 1>, ...)
  [ throws <type d'exception>] {
  // declaration
  }

# Déclaration : méthodes

- int foo() {
  return 0;
  }

- void bar(int a, int b) {
  if(a < b) then return a;
  else return b;
  }

# Déclarations : méthodes

- void caramba() throws Exception;

# Déclaration : constructeurs

- <nom de classe>(<type de arg 1> <nom de arg 1>, ...)
- Voiture(String immatriculation, int nombrePassagers)

# Déclaration : classes

```
class <nom de classe> {
  <variables de classe>
  <variables d'instance>
  <methodes de classe>
  <methodes d'instance>
}
```

# Déclaration : classes

```
class Voiture {
  String immat;
  int nbMaxPas;
  int passagersActuels;

  Voiture() {
    immat = "Pas encore immatriculée";
    nbMaxPas = 0;
  }
```

# Déclaration : classes

(suite)

```
 boolean charger(int passagers) {
   boolean reussi;
   if(passagersActuels+passagers>nbMaxPas)
     reussi = false;
   else {
     passagersActuels = passagersActuels + passagers;
     reussi = true;
   }
   return reussi;
 }
}
```

# Déclaration : packages

- Les packages en Java sont très importants.
- Ils définissent des “librairies”
- Ils ressemblent à des chemins d’accès

# Déclaration : packages

- Un exemple : java.util
- La classe Date est définie dans ce package
- Son chemin est donc : java.util.Date

# Déclaration : packages

- Un objet de type Date peut être déclaré de deux façons différentes:
  - java.util.Date maDate = new java.util.Date();
  - import java.util.Date; // en début de fichier
    (...)
    Date maDate = new Date();

# Déclaration : packages

- On peut combiner les “import”:
  import java.util.*;
- Attention, si vous avez des classes portant le même nom dans des packages différents, le dernier chargé prévaut.

# Déclaration : packages

- package <chemin du package>;
  // déclarations de classes

- Note : le “chemin” n’est pas nécessairement limité aux morceaux existants
  Ex: package mon.super.package.dont.je.ne.pense.que.du.bien; // est un package valide

# Petits détails cosmétiques

- Un nom de classe commence par une Majuscule, et chaque mot composant son nom aussi : MaSuperClasse
- Sauf cas exceptionnel, chaque classe est définie dans un fichier portant son nom
- Les constantes de classe sont en MAJUSCULES

# Petits détails cosmétiques

Les noms de méthodes commencent par une minuscule, et chaque mot les composant par une majuscule
Ex : maSuperMethode()

# Point d'entrée

# En C

- int main(void)
- int main(int argc, char** argv)

# En Java

- Chaque classe peut contenir un point d'entrée
- La machine virtuelle cherche d'abord la classe requise puis la méthode

# Le monstre

- public static void main(String args[])
- Forme obligatoire

# Le monstre

- public static void main(String args[])
- Accessible par tous (et surtout par la machine virtuelle)

# Le monstre

- public static void main(String args[])
- Indépendant de l'instance dans laquelle on se trouve (pas besoin de faire un new)

# Le monstre

- public static void main(String args[])
- Pas de return dans le main en Java. On utilise exit(int status)

# Le monstre

- public static void main(String args[])
- String est un objet qui a des méthodes, pas un tableau de caractères

# Le monstre

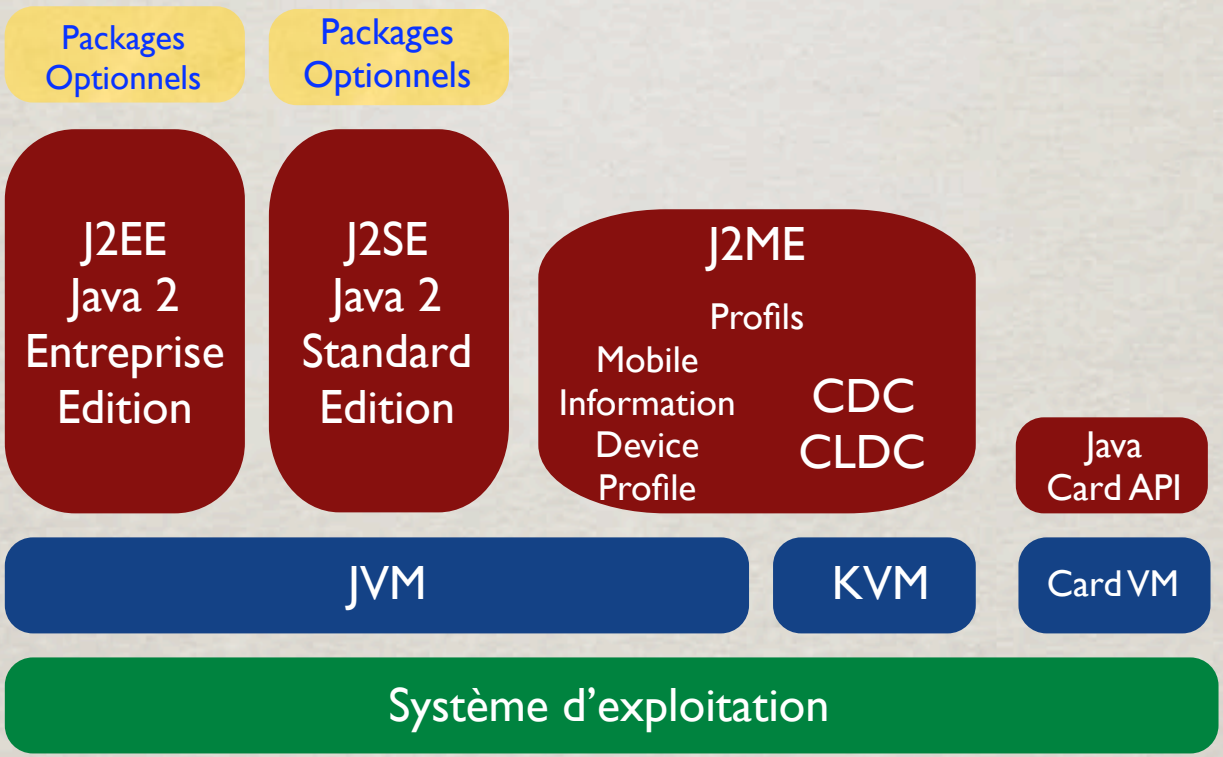
- public static void main(String args[])
- Le tableau est une entité ambigüe en Java: c'est un objet, mais sa syntaxe ressemble à celle du C

# Parenthèse sur la machine virtuelle

# LA JVM

- La machine virtuelle est une application, un processus
- Elle se lance, puis charge la classe principale, puis les références, puis les references des références, etc...

# Les différentes VMs



# JVM utilisée

- J2SE 1.4.2 (ou 1.5 mais seulement les bouts compatibles)
- Plate-formes : Windows, Unices ou Mac OS X
- IDE : Eclipse, NetBeans, CodeWarrior, Xcode, emacs, vim/javac, ...