

# Programmation Orientée Objet - Chapitre 1 Introduction et concepts de la POO

Jean-Marie Normand  
`jean-marie.normand@ec-nantes.fr`  
Bâtiment E - Bureau 211

# Objectifs du cours

Concevoir et écrire des applications en utilisant un langage orienté objet : Java Nous allons donc voir dans ce module :

- le concept d'objet et les notions associées
- en s'appuyant sur ce qui a été fait en algorithmique et programmation
- toutes ces notions et tous ces concepts seront illustrés dans un nouveau langage de programmation : Java

# Plan du cours I

- 1 Introduction
- 2 Rappels ALGPR
- 3 Introduction à Java
- 4 Concepts de la Programmation Objet : Objets, Classes

# Plan

## 1 Introduction

# Contexte général

## Informatique de l'**ingénieur**, pas de l'informaticien

- savoir utiliser ce qui a été fait par d'autres
  - ▶ comprendre ce qui est proposé par des composants logiciels existants
  - ▶ savoir lire, comprendre et écrire la documentation
- travailler en (petite) équipe :
  - ▶ comprendre ce qu'ont fait les autres
  - ▶ être capable de fournir/transmettre/expliquer son travail aux autres

# Objectifs de l'enseignement

## ■ Analyse et conception orientée-objet

- ▶ différences avec l'approche impérative/procédurale (vue en 1<sup>re</sup> année)
- ▶ comprendre les concepts de la programmation orientée objet (POO)
- ▶ un TP "projet" que vous suivrez tout au long du module
- ▶ développement d'application simple
- ▶ pré-requis pour BEAUCOUP d'autres cours de votre option !

# Objectifs de l'enseignement

## ■ Analyse et conception orientée-objet

- ▶ différences avec l'approche impérative/procédurale (vue en 1<sup>re</sup> année)
- ▶ comprendre les concepts de la programmation orientée objet (POO)
- ▶ un TP "projet" que vous suivrez tout au long du module
- ▶ développement d'application simple
- ▶ pré-requis pour BEAUCOUP d'autres cours de votre option !

## ■ Programmation avec un (nouveau) langage à objets : **Java** !

- ▶ utilisation de bibliothèques existantes : API Java
- ▶ utilisation d'outils de développement (IDE) modernes : NetBeans

# Objectifs de l'enseignement

- Analyse et conception orientée-objet
  - ▶ différences avec l'approche impérative/procédurale (vue en 1<sup>re</sup> année)
  - ▶ comprendre les concepts de la programmation orientée objet (POO)
  - ▶ un TP “projet” que vous suivrez tout au long du module
  - ▶ développement d'application simple
  - ▶ pré-requis pour BEAUCOUP d'autres cours de votre option !
- Programmation avec un (nouveau) langage à objets : **Java** !
  - ▶ utilisation de bibliothèques existantes : API Java
  - ▶ utilisation d'outils de développement (IDE) modernes : NetBeans
- Un lien avec l'industrie
  - ▶ Java est un langage très utilisé : applications Web, applications “lourdes”, etc.
  - ▶ Java est multi-plateforme : il s'abstrait des contraintes matérielles ⇒ on ne développe qu'une fois pour tous les environnements (Windows, Linux, Mac)



# Modalités

## ■ Répartition du volume horaire (32h)

- ▶ 10h cours
- ▶ 20h TD/TP (sur machine)
- ▶ 2h DS

## ■ Intervenants

- ▶ Jean-Marie Normand (option Informatique)
- ▶ Jean-Yves Martin (option Informatique)
- ▶ Morgan Magnin (option Informatique)

## ■ Évaluation


- ▶ EVI (2/3) : DS sur table (90%) + moyenne des QCM (10%, voir plus bas)
- ▶ EVC (1/3) : évaluation des rendus liés au projet (chaque séance de TP)

# Modalités pédagogiques

## ■ QCM

- ▶ début de chaque TD/TP, 5 minutes maximum
- ▶ questions simples portant sur le CM précédent et/ou à préparer à l'avance, avec le barème suivant :
  - bon : +1 point
  - **faux : -1 point !!!**
  - rien : 0 point
- ▶ absence (non excusée) aux QCMs : pénalité sur la note totale de QCM

## ■ Supports de cours :

- ▶ code des exemples (et plus) sur <https://hippocampus.ec-nantes.fr>
- ▶ dans la version électronique de ce document, le symbole  renvoie à un fichier contenant le code
- ▶ vidéos complémentaires aux CMs et **nouveau cette année : vous serez censés les avoir vues !**

# Modalités pédagogiques

## ■ TP

- ▶ notés sur 6 critères
- ▶ note pour chaque critère: de A à D ou F
- ▶ pondérations différentes suivant les TPs
- ▶ Critères :
  - Qualité du rapport (intro, conclu, phrases, grammaire, etc.)
  - Qualité du code : respect des conventions, respect des droits d'accès, accepteurs, getters, attributs, etc.
  - Mise en évidence des résultats obtenus : tests, vérification des résultats obtenus, justifications, éventuellement graphes etc.
  - Respect du sujet et des consignes.
  - Progression/Avancement par rapport au sujet.
  - Critère spécifique à chaque TP : compréhension de la notion présentée

# Projet

## RPG simpliste

Nous construirons ensemble à partir de zéro un jeu de rôle (ou *role-playing game*, RPG) en mode texte simpliste. Ce projet sera poursuivi tout au long du module et servira de base d'application des notions vues en cours. Déroulement :

- une première séance de prise en main de l'environnement de développement utilisé tout au long de l'année : NetBeans
- des slides pour chaque séance : contenant le travail à faire, des aides et les exercices
- un rapport à rendre à la fin de chaque séance de TP avec les réponses à chaque exercice
- chaque rapport sera noté !

# Plan global du cours

- Rappels ALGPR
- Concepts Programmation Orientée Objet
- Héritage et polymorphisme
- Structures de Données
- Abstraction et interfaces
- Exceptions
- Optionnel (disponible sur le serveur pédagogique) : Threads
- Optionnel (disponible sur le serveur pédagogique) : Interfaces Graphiques Utilisateur (GUI) avec SWING

# Plan

- 2 Rappels ALGPR
  - D'AGLPR à la Programmation Orientée Objet
  - L'algorithmique
  - Programmation C++

# D'ALGPR (programmation procédurale – PP) à la Programmation Orientée Objet (POO) ?

## ■ Points communs :

- ▶ séparation données - traitements
  - toujours des algorithmes
  - toujours des fonctions
- ▶ syntaxe proche du C(++)

## ■ Différences :

- ▶ raisonnement différent :
  - la **PP** repose sur des fonctions que l'on appelle avec des données pour résoudre un problème
  - la **POO** s'appuie sur les données à qui l'on demande de réaliser des tâches (la donnée est au cœur de la POO)
- ▶ utilisation de ressources existantes (bibliothèques, etc.)
- ▶ **nouveau** langage de programmation : Java

# Programmation procédurale

- Analyse descendante
- Fonctions sont au cœur de l'analyse
- Conception :
  - ▶ définir les structures de données
  - ▶ définir les traitements
  - ▶ programme principal = enchaînement des traitements

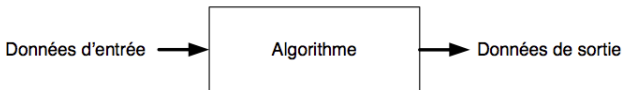


Figure: Principe de la programmation procédurale.



# Rappels sur ALGPR

## Tout ALGPR en quelques minutes...

- Les principes de l'algorithmique : comment écrire des algorithmes (indépendamment de tout langage de programmation)
  - ▶ Structures de contrôle : structures conditionnelles, structures itératives
  - ▶ Représentation des données : variables, constantes, structures de données (tableaux, types complexes, listes, arbres)
  - ▶ Fonctions et procédures
  - ▶ Récursivité

# Rappels sur ALGPR

## Tout ALGPR en quelques minutes...

- Les principes de l'algorithmique : comment écrire des algorithmes (indépendamment de tout langage de programmation)
  - ▶ Structures de contrôle : structures conditionnelles, structures itératives
  - ▶ Représentation des données : variables, constantes, structures de données (tableaux, types complexes, listes, arbres)
  - ▶ Fonctions et procédures
  - ▶ Récursivité
- Analyse descendante : comment décomposer un problème complexe en sous problèmes plus simples pouvant être résolus facilement

# Rappels sur ALGPR

## Tout ALGPR en quelques minutes...

- Les principes de l'algorithmique : comment écrire des algorithmes (indépendamment de tout langage de programmation)
  - ▶ Structures de contrôle : structures conditionnelles, structures itératives
  - ▶ Représentation des données : variables, constantes, structures de données (tableaux, types complexes, listes, arbres)
  - ▶ Fonctions et procédures
  - ▶ Récursivité
- Analyse descendante : comment décomposer un problème complexe en sous problèmes plus simples pouvant être résolus facilement
- Programmation impérative (procédurale) en C++ (sans objet) :
  - ▶ Traduction des structures algorithmiques en C++
  - ▶ Fichiers d'entêtes (.h/.hpp) et de programmes (.c/.cpp)
  - ▶ Écriture de sous-programmes
  - ▶ Utilisation de bibliothèques standard C++

# Algorithme

## Définition (cf. cours ALGPR)

Une séquence d'instructions qui décrit, pour un environnement donné, les étapes à suivre pour obtenir un résultat (de façon reproductible) à partir de données.

Éléments essentiels pour un algorithme :

- données : variables, constantes, structures de données
- opérateurs : mathématiques, logiques, comparaisons, affectation, etc.
- structures de contrôle : si/alors/sinon, boucle pour, boucle tantque
- fonctions

## Exemple d'algorithme (1)

---

### Algorithm 1 Algorithme mystère : que fais-je ?

---

```
1: nombre ← 0
2: tant que nombre < 10 ou nombre > 20 faire
3:   nombre ← lire("Entrez un nombre entre 10 et 20 svp !")
4:   si nombre > 20 alors
5:     écrire "Entrez un nombre plus grand !"
6:   sinon
7:     si (nombre < 10) alors
8:       écrire "Entrez un nombre plus petit !"
9:     fin si
10:  fin si
11: fin tant que
12: écrire "Vous avez entré le nombre : ", nombre
```

## Exemple d'algorithme (1)

Votons à mains levées !

### Solution 1

- ❶ Demande un nombre entre 10 et 20.
- ❷ Affiche une aide **correcte** pour l'utilisateur si le nombre est  $<10$  ou  $>20$ .
- ❸ Continue à demander tant que le nombre est  $<10$  **OU**  $>20$ .
- ❹ Affiche le nombre entré par l'utilisateur.

### Solution 2

- ❶ Demande un nombre entre 10 et 20.
- ❷ Affiche une aide **incorrecte** pour l'utilisateur si le nombre est  $<10$  ou  $>20$ .
- ❸ Continue à demander tant que le nombre est  $<10$  **ET**  $>20$ .
- ❹ Affiche le nombre entré par l'utilisateur.

### Solution 3

- ❶ Demande un nombre entre 10 et 20.
- ❷ Affiche une aide **correcte** pour l'utilisateur si le nombre est  $<10$  ou  $>20$ .
- ❸ Continue à demander tant que le nombre est  $>10$  **OU**  $<20$ .
- ❹ Affiche le nombre entré par l'utilisateur.

## Exemple d'algorithme (2)

### fonctionMystère

**Algorithm 2** Fonction mystère : que fais-je ?

**Entrées :** caractère *carC*, caractère *carP*, entier *t*

**Sorties :** entier *t*

- 1: si *carP* = 'l' et *carC* = 'e' alors
- 2:    $t \leftarrow t + 1$
- 3: fin si
- 4: retourner *t*

## Exemple d'algorithme (2)

Votons à mains levées !

### Solution 1

- Compte le nombre de fois qu'un 'l' est suivi d'un 'e'.

### Solution 2

- Compte le nombre de fois qu'un 'e' est suivi d'un 'l'.

### Solution 3

- On ne peut pas savoir si on compte les 'le' ou les 'el'.

### Solution 4

- Je n'en sais rien.



## Exemple d'algorithme (3)

---

### Algorithm 4 Algorithme mystère (2) : que fais-je ?

---

```
1: car1 ← ""
2: car2 ← ""
3: nb ← 0
4: car1 ← lire("Entrez un caractère svp !")
5: tant que car1 ≠ '.' faire
6:   nb ← fonctionMystère(car1, car2, nb)
7:   car2 ← car1
8:   car1 ← lire("Entrez un caractère svp !")
9: fin tant que
10: écrire "Le nombre de motif est de : ", nb
```

## Exemple d'algorithme (3)

Votons à mains levées !

### Solution 1

- 1 Compte le nombre de fois qu'un 'l' est suivi d'un 'e'.

### Solution 2

- 1 Compte le nombre de fois qu'un 'e' est suivi d'un 'l'.

### Solution 3

- 1 Je n'en sais rien.

# Programmation C++

## Principe général

- Objectif : production de **code source**
- Comment ? : **Traduction** de l'algorithme
- Outil ? : Langage de programmation **C++**

## Spécificités C/C++

- Déclaration des fonctions dans les fichiers d'entête (.h/.hpp)
- Écriture du code dans les fichiers .cpp
- Fonctions auxiliaires ne peuvent retourner plusieurs valeurs (contrairement aux fonctions auxiliaires des algorithmes)
- Nécessité d'une fonction principale (**main**) qui correspond à l'algorithme

# Traduction d'un algorithme en C++

```
#include <stdio.h>
#include <stdlib.h>
using namespace std;
// Algorithme mystere 1 traduit en C++
int main(int argc, char** argv) {
    int nombre = 0;
    while(nombre < 10 or nombre > 20) {
        cout << "Entrez un nombre entre 10 et 20 svp!" << endl;
        cin >> nombre;
        if(nombre > 20) {
            cout << "Entrez un nombre plus petit !" << endl;
        }
        else {
            if(nombre < 10) {
                cout << "Entrez un nombre plus grand !" << endl;
            }
        }
    }
    cout << "Vous avez entre le nombre : " << nombre << endl;
}
```

# Traduction d'une fonction en C++

## Fichier mystere.h

```
// Declaration de la fonction mystere
int mystere(char carC, char carP, int t);
```

## Fichier mystere.c

```
// Fonction mystere traduite en C++
int mystere(char carC, char carP, int t) {
    if(carP == 'l' && carC == 'e') {
        t = t+1;
    }
    return t;
}
```

# Algo principal avec la fonction mystère en C++

```
#include <stdio.h>
#include <stdlib.h>
#include "mystere.h"
using namespace std;
// Algorithme traduit en C++
int main(int argc, char** argv) {
    // Definition et initialisation des variables
    char car1 = '';
    char car2 = '';
    int nb = 0;
    // Recuperation caractere
    cout << "Entrez un caractere svp" << endl;
    cin >> car1;
    while(car1 != '.') {
        nb = mystere(car1,car2,nb);
        // Echange des caracteres
        car2 = car1; cout << "Entrez un caractere" << endl; cin >> car1;
    }
    cout << "Le nombre de motif est de : " << nb << endl;
}
```

# Plan

## 3 Introduction à Java

# Historique de Java

- 1990-1992 : projet Oak chez Sun Microsystems : lancé par des ingénieurs mécontents de C++
- 1995 : Java 1.0 lancé par Sun Microsystems
  - ▶ James Gosling considéré comme son “père”
  - ▶ promesse : “*Write Once, Run Anywhere*” (WORA)
  - ▶ sur les bases de C
  - ▶ ayant pour but d’être intégré dans tout type d’appareils (électroménager, etc.)
  - ▶ en plein essor d’Internet :
    - succès des *applets* (programmes côté client)
    - succès des *servlets* (programmes côté serveur)
- 2009 : acquisition par Oracle



# Particularités de Java (1)

- Suppression de certaines sources de confusion de C(++) :
  - ▶ plus de structures (pas de `typedef struct`)
  - ▶ plus de macros (pas de `#define`)
  - ▶ plus de **pointeurs** (apparents) !
  - ▶ simplification du mécanisme d'importation de fichiers (plus puissant et pratique que les `#include`)  $\Rightarrow$  `import`
  - ▶ disparition du mécanisme d'édition de liens
  - ▶ langage fortement typé

## Particularités de Java (2)

- Gestion autonome de la mémoire (mécanisme du **ramasse-miettes** ou *Garbage Collecting - GC*) :
- Portable et multi-plateforme
- Large variété d'applications potentielles : “desktop”, web (client/serveur), embarqué, mobile, cartes à puces, etc.
- Nombreuses bibliothèques disponibles (via son API ou *Application Programming Interface*)
- Pour information : évolution du nombre de classes documentées dans l'API Java par version du JDK

Version	1.0(95)	1.1(97)	1.2(98)	1.3(00)	1.4(02)	5.0(04)	6(06)	7(11)	8(14)	...	12 (19)
Classes	212	504	1520	1842	2991	3279	3793	4024	4240	...	4433
Packages	8	23	59	76	135	166	203	209	217	...	225

Figure: Évolution du JDK.

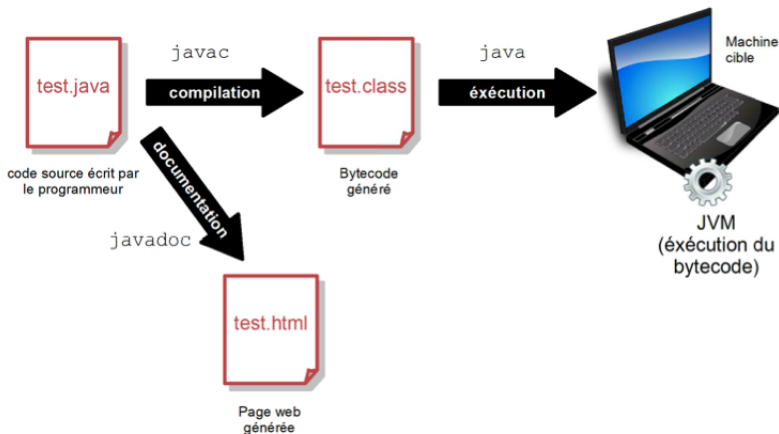
# Java

- Un environnement de développement complet et gratuit !
- Le **JDK** (*Java Development Kit*) englobe :
  - ▶ **JRE** : *Java Runtime Environment*, permet l'exécution des applications Java et contient :
    - **JVM** : *Java Virtual Machine* : permet la simulation d'une véritable machine physique
  - ▶ **API** (*Application Programming Interface*) : contenant l'ensemble des fonctionnalités de la bibliothèque standard de Java
  - ▶ compilateur : **javac**
  - ▶ générateur automatique de documentation : **javadoc**
  - ▶ débogueur : **jdb**

# Étapes du développement en Java

- ❶ Création du code source et écriture de la documentation
  - ▶ Outil : éditeur de texte (JEdit, gEdit, etc.), environnement de développement intégré (IDE - *Integrated Development Environment*) Eclipse, [NetBeans](#), etc.
- ❷ Compilation du code source en *bytecode* (celui de la JVM) et génération de la documentation
  - ▶ Outils : compilateur Java (javac), générateur de documentation (javadoc)
- ❸ Déploiement du bytecode sur la machine cible
  - ▶ Outils : réseau, disques, clé USB, etc.
- ❹ Chargement de l'application et exécution sur la machine cible
  - ▶ Outil : machine virtuelle Java (JVM)

# Étapes du développement en Java



# Ressources en ligne Java

- Site officiel Java : <http://www.java.com/fr/>
- Tutoriels en ligne : <http://docs.oracle.com/javase/tutorial>
- **Documentation en ligne - JavaDoc :**  
<http://docs.oracle.com/javase/8/docs>

# Exemple JavaDoc (1)

Overview	Package	Class	Use	Tree	Deprecated	Index	Help	Java™ Platform Standard Ed. 7
Prev Class	Next Class	Frames	No Frames	All Classes				
Summary: Nested	Field	Constructor	Method	Details	Field	Constructor	Method	
<pre>java.util  Class ArrayList&lt;E&gt;  java.lang.Object   java.util.AbstractCollection&lt;E&gt;     java.util.AbstractList&lt;E&gt;       java.util.ArrayList&lt;E&gt;  All Implemented Interfaces:   Serializable, Cloneable, Iterable&lt;E&gt;, Collection&lt;E&gt;, List&lt;E&gt;, RandomAccess  Direct Known Subclasses:   AttributeList, RoleList, RoleUnreadList   public class ArrayList&lt;E&gt;     extends AbstractList&lt;E&gt;     implements List&lt;E&gt;, RandomAccess, Cloneable, Serializable  Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list. (This class is roughly equivalent to Vector, except that it is unsynchronized.) The list, Iterator, get, set, Iterator, and ListIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation. Each ArrayList instance has a capacity. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost. An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.  Note that this implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly replaces the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the Collections.synchronizedList method. This is best done at creation time, to prevent accidental unsynchronized access to the list.  List list = Collections.synchronizedList(new ArrayList&lt;&gt;());  The iterators returned by this class's iterator and ListIterator methods are fail-fast: if the list is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove or add methods, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than making arbitrary, non-deterministic behavior at an undetermined time in the future.  Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.  This class is a member of the Java Collections Framework.  Since:   1.2  See Also:   Collection, List, LinkedList, Vector, Serializable Form</pre>								

Figure: Exemple de documentation générée automatiquement par la Javadoc

## Exemple JavaDoc (2)

### Field Summary

#### Fields inherited from class java.util.AbstractList

modCount

### Constructor Summary

#### Constructors

##### Constructor and Description

`ArrayList()`

Constructs an empty list with an initial capacity of ten.

`ArrayList(Collection<E> extends E? c)`

Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.

`ArrayList(int initialCapacity)`

Constructs an empty list with the specified initial capacity.

### Method Summary

#### Methods

Modifier and Type	Method and Description
boolean	<code>add(E e)</code> Appends the specified element to the end of this list.
void	<code>add(int index, E element)</code> Inserts the specified element at the specified position in this list.
boolean	<code>addAll(Collection&lt;E&gt; extends E? c)</code> Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator.
boolean	<code>addAll(int index, Collection&lt;E&gt; extends E? c)</code> Inserts all of the elements in the specified collection into this list, starting at the specified position.
void	<code>clear()</code> Removes all of the elements from this list.
Object	<code>clone()</code> Returns a shallow copy of this <code>ArrayList</code> instance.
boolean	<code>contains(Object o)</code> Returns true if this list contains the specified element.
void	<code>ensureCapacity(int minCapacity)</code> Increases the capacity of this <code>ArrayList</code> instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
E	<code>get(int index)</code> Returns the element at the specified position in this list.
int	<code>indexOf(Object o)</code> Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	<code>isEmpty()</code> Returns true if this list contains no elements.



## Exemple Javadoc (3)

### add

```
public boolean add(E e)
```

Appends the specified element to the end of this list.

**Specified by:**

add in interface `Collection<E>`

**Specified by:**

add in interface `List<E>`

**Overrides:**

add in class `AbstractList<E>`

**Parameters:**

e - element to be appended to this list

**Returns:**

true (as specified by `Collection.add(E)`)

**Figure:** Exemple de documentation générée automatiquement par la Javadoc : détail d'une méthode.

# JavaDoc

## Lisez la JavaDoc !!!

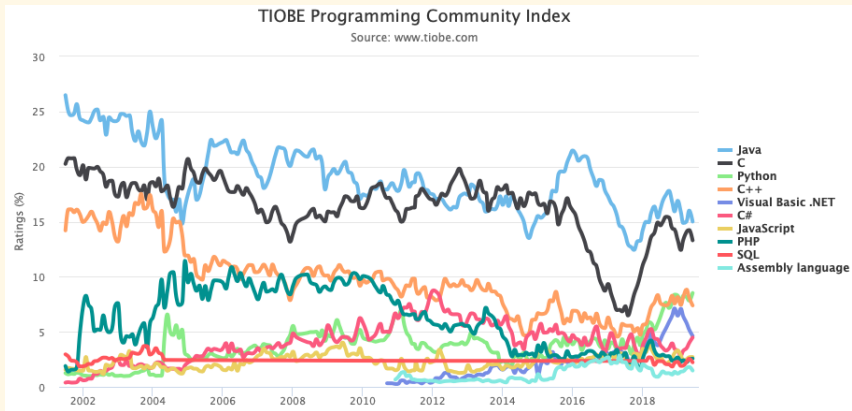
Il existe énormément de classes en Java, il est impossible de les connaître toutes sur le bout des doigts. Il faut donc **très souvent se référer à la JavaDoc** pour aller chercher les informations nécessaires.

Quelques classes utiles à connaître en Java :

- ArrayList
- LinkedList
- String
- HashMap
- Date
- System
- etc.

# Popularité de Java

Java est premier (de peu) !



## Popularité de Java (2)

Java est premier (de peu) !

Jun 2019	Jun 2018	Change	Programming Language	Ratings	Change
1	1		Java	15.004%	-0.36%
2	2		C	13.300%	-1.64%
3	4	▲	Python	8.530%	+2.77%
4	3	▼	C++	7.384%	-0.95%
5	6	▲	Visual Basic .NET	4.624%	+0.86%
6	5	▼	C#	4.483%	+0.17%
7	8	▲	JavaScript	2.716%	+0.22%
8	7	▼	PHP	2.567%	-0.31%
9	9		SQL	2.224%	-0.12%
10	16	▲▲	Assembly language	1.479%	+0.56%

Figure: Indicateur TIOBE de la popularité des langages de programmation

# Programmation en Java

## Java est un langage orienté objet

On peut manipuler des données (variables, etc.) différentes :

- des **types de “base”** (ou **types primitifs**) :

- ▶ `int`
- ▶ `float`
- ▶ `double`
- ▶ `char`
- ▶ `boolean`

- des objets

- des tableaux

- etc.

# Mon premier programme Java (1)

```
// Ce code source doit se trouver dans un fichier appele
// 'PremierProg.java'
public class PremierProg {

    // Cette fonction represente le programme principal
    // elle equivaut a la fonction main() de C/C++
    public static void main(String[] args)
    {
        // cet appel equivaut au std::cout de C++
        System.out.println("Hello World!");
    }
}
```

## Mon premier programme Java (2)

Comment le compile t'on ? L'exécute t'on ?

### “Old school” : dans un terminal

- Compilation :  
`javac PremierProg.java`
- Exécution :  
`java PremierProg`
- Résultat :  
Hello World!

### IDE moderne (cf. TP1)

- Vérifie que l'éditeur n'affiche pas d'icône d'erreur
- Clic sur le bouton “Run” !!
- Résultat :  
Hello World!

# Mon premier programme Java (3)

Explications :

- vous comprendrez mieux à l'issue de la première séance de cours !
- la fonction principale appartient forcément à une **classe**
- le nom du fichier **.java** doit être le même que celui de la classe (par convention)
- l'entête de la fonction **main** est **toujours** le même !
- **String[]** représente un tableau statique (**[]**) de chaînes de caractères (la classe **String** dont on reparlera en TP)



# Plan

## 4 Concepts de la Programmation Objet : Objets, Classes

# Programmation procédurale

- Centrée sur les **procédures** (fonctions, opérations) :
  - ▶ décomposition des fonctionnalités d'un programme en procédures qui vont s'exécuter séquentiellement (éventuellement récursivement)
- Découplage données
  - ▶ les données sont indépendantes procédures
  - ▶ les données à traiter sont passées en arguments aux procédures
  - ▶ absence de sémantique entre données (variables) et traitements (fonctions/procédures)
- Tend à générer du code "Spaghetti"
  - ▶ la maintenance et l'ajout de nouvelles fonctionnalités demandent de modifier ou d'insérer des séquences dans ce qui existe déjà
  - ▶ peut devenir complexe très rapidement
  - ▶ modularité et abstraction absente (ou presque)
  - ▶ réutilisation ardue ⇒ "Couper-coller" = DANGER!
  - ▶ travail d'équipe difficile (peu modulaire), donc la qualité du code en souffre

# Programmation procédurale

## Workflow

- définir les structures de données et les données (**variables**),
- définir les traitements associés aux données (**fonctions**),
- écrire un **algorithme principal** qui va enchaîner les traitements sur les données.

## Manipulation de données

- types simples : entiers, flottants, booléens, etc.
- possibilité d'utiliser des :
  - ▶ structures : regroupement de types au sein d'une entité
  - ▶ tableaux : regroupement de données identiques (de même type simple ou structure)

# Programmation Orientée Objet (POO)

## POO

- centrée sur les **données**
- organisation de programmes complexes
- regroupement sémantique entre données et traitements associés

## Workflow

- définir et faire interagir des briques logicielles appelées “**objets**”,
- un objet est une représentation informatisée d'une entité (tangible ou intangible) du monde réel,
- en faisant communiquer les objets, ils collaborent dans un but précis (à déterminer).

# Programmation procédurale vs. orientée objet

## Programmation Procédurale :

- Procédure (fonction) = traitement qui peut être décomposé en sous-traitements jusqu'à obtention de traitements basiques (**analyse descendante**)
- travaille sur l'**action**, le **verbe**

## Calcul de la vitesse d'une voiture

`calculVitesse(voiture)`

⇒ le code faisant référence à une **voiture** est "**fondue**", **dispersé** dans l'ensemble des fonctions du programme !

## Programmation Orientée-Objet :

- manipule uniquement des **objets** : regroupement de **variables** et de **traitements** au sein d'une **même entité**
- le **sujet** est prépondérant !!!

## Calcul de la vitesse d'une voiture

`voiture.calculVitesse()`

⇒ tout le code se rapportant à l'**objet voiture** est **regroupé** au sein de la classe !

# Bonnes propriétés de la POO

La POO présente un certain nombre de bonnes propriétés :

- clarté conceptuelle
- maintenabilité
- modularité
- robustesse face aux modifications
- lisibilité

# Concepts de la POO

Fondamentaux :

- objet, classe et instance
- attributs d'instance et attributs de classe
- méthodes d'instance et méthodes de classe
- constructeur
- encapsulation
- héritage
- polymorphisme
- abstraction

# Qu'est-ce qu'un objet ?

Un objet :

- **modélise** toute entité identifiable, concrète ou abstraite, manipulée par une application logicielle
  - ▶ une chose tangible : une ville, un étudiant, un bouton sur l'écran, etc.
  - ▶ une chose conceptuelle : une date, une réunion, un planning de réservation, etc.
- **réagit** à certains messages qu'on lui envoie de l'extérieur ; la réaction détermine le **comportement** de l'objet
- ne réagit pas toujours de la même façon à un même message ; sa réaction dépend de l'**état dans lequel il se trouve**



# Un objet possède :

- une **identité unique** (pour distinguer un objet d'un autre)
- un **état interne** donné par la valeur d'un certain nombre de variables qu'on appelle des **attributs** :
  - ▶ l'ensemble des attributs décrit l'**état** de l'objet à un instant donné (cette personne mesure 1,75m, pèse 54 kg et s'appelle Charles)
  - ▶ les attributs sont typés et nommés (attribut *taille* de type réel)
- un **comportement** (capacités d'action de l'objet) donné par des fonctions qu'on appelle des **méthodes** (procédures ou fonctions)
  - ▶ les méthodes définissent ce que l'objet peut faire et comment il peut le faire

# Notion d'encapsulation

- regrouper dans le même **objet** les **données** et les **traitements** qui lui sont spécifiques
  - ▶ **attributs** : données incluses dans un objet
  - ▶ **méthodes** : fonctions et/ou procédures (traitements) définies dans un objet
- isolement et dissimulation des **détails d'implémentation**
- notion d'“*interface*” d'un objet :
  - ▶ ce que l'utilisateur (le programmeur-utilisateur) peut utiliser de l'objet
  - ▶ empêche l'accès aux données par un autre moyen que les services proposés

## Définition d'un objet

Un objet est complètement défini par ses **attributs** et ses **méthodes**

# Principe d'encapsulation

- L'accès aux données (attributs) de l'objet ne devrait être fait qu'au travers des méthodes
  - ▶ les données (attributs) sont **privées** (cachées)
  - ▶ les méthodes **publiques** définissent l'*interface* de l'objet
- Intérêt : la modification des structures de données n'affecte pas les programmeurs-utilisateurs qui utiliseront l'objet (via l'interface)
  - ▶ masquage de l'implémentation : robustesse du code
  - ▶ facilité d'évolution du logiciel
  - ▶ pas de modification de l'interface

# Illustration de l'encapsulation

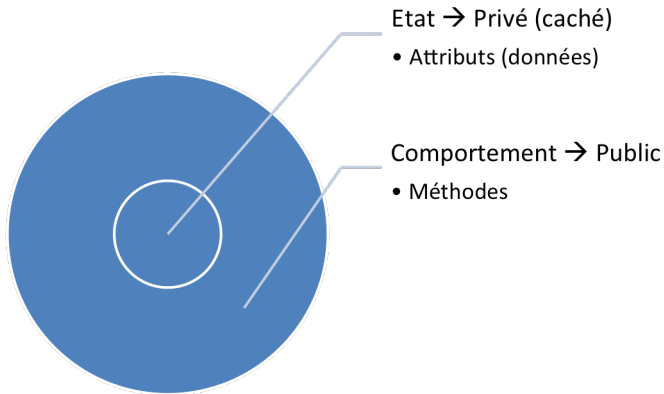


Figure: Concept d'encapsulation.

## Exemple concret d'encapsulation : Voiture

Une voiture possède des **attributs** :

- constructeur
- modèle
- moteur
- couleur
- puissance
- etc.

Et des **méthodes** :

- démarrer
- avancer
- changer de vitesse
- reculer
- s'arrêter
- etc.

## Exemple concret d'encapsulation : Voiture

Une voiture possède des **attributs** :

- constructeur
- modèle
- moteur
- couleur
- puissance
- etc.

Et des **méthodes** :

- démarrer
- avancer
- changer de vitesse
- reculer
- s'arrêter
- etc.

### Encapsulation

Si l'on modifie le type de moteur d'une voiture (e.g. essence  $\Rightarrow$  diesel) :

- la voiture **garde les mêmes fonctionnalités**
- **même si** à l'intérieur (sous le capot) elles ne sont pas implémentées de la même manière !

# Pourquoi de l'encapsulation ?

L'encapsulation permet de :

- regrouper les données et les traitements
- définir **deux niveaux** de perception des objets :
  - ▶ niveau “**externe**” = l’“interface” : partie **publique** “visible” et utilisable par les programmeurs-utilisateurs
  - ▶ niveau “**interne**” : détails d’implémentation, partie **privée** connue uniquement des programmeurs de la classe
- tant qu’on ne touche pas à l’interface, il est possible :
  - ▶ **d’améliorer les performances** d’une classe en modifiant, optimisant le niveau “interne”
  - ▶ sans impacter l’utilisation (via l’interface) de la classe par les programmeurs-utilisateurs de la classe !
- p. ex. si j’améliore les performances de la classe **Voiture** j’optimise également le comportement de toutes les classes qui utilisent la classe **Voiture** **sans qu’elles s’en rendent compte**

## Vers la notion de classe

### Bilan de l'encapsulation :

- constat : les objets d'une même **famille** possèdent les mêmes caractéristiques (**mêmes attributs et mêmes méthodes**) :
  - ▶ toutes les voitures ont un constructeur, une couleur, peuvent démarrer, s'arrêter, avancer ou reculer
- une **classe d'objets** (ou simplement **classe**) caractérise un ensemble d'objets semblables
  - ▶ c'est la classe des voitures !
- un objet donné est alors vu comme une occurrence (une **instance**) d'une classe
  - ▶ ma voiture est une **instance** de la classe des voitures

### Les objets associés à une classe se nomment instances

Une **instance** est un objet, occurrence d'une classe, qui possède la structure définie par la classe (les **attributs**) et sur lequel les opérations



# Classes et instances

Les objets (instances) sont créés (instanciés) à partir de “moules” que l’on appelle des **classes** :

- **classe** = schéma, moule, modèle d’objets décrivant :
  - ▶ partie **privée** : structure de données interne (**attributs**), **certaines méthodes** servant à maintenir l’intégrité de l’objet
  - ▶ partie **publique** (interface) : méthodes utilisables par les programmeurs-utilisateurs
- la classe est un générateur d’objet : par **instanciation**, on peut fabriquer des objets (instances) respectant ce schéma/moule/modèle

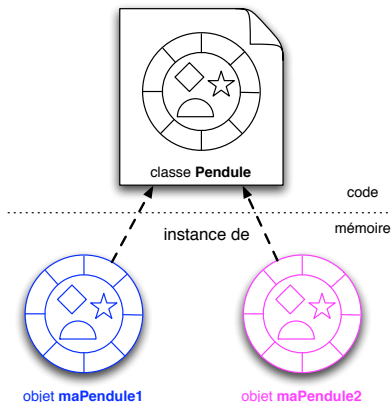
# Vue duale

Mêmes choses, mais vues des classes :

- une **classe** est un **modèle** de définition pour des objets
  - ▶ ayant même structure (même ensemble d'attributs)
  - ▶ ayant même comportement (mêmes opérations, méthodes)
  - ▶ ayant une sémantique commune
- les **objets** sont des représentations **dynamiques** et concrètes "vivantes" du modèle défini pour eux au travers de la classe
  - ▶ une classe permet d'**instancier** (créer) plusieurs objets
  - ▶ chaque objet est **instance** d'une (seule) classe

# Classes et instances

Exemple : la classe Pendule est le moule commun des deux objets



# Intérêt des classes

- Les classes permettent la réutilisation de code :
  - ▶ on définit une classe **une fois pour toute**
  - ▶ on instancie cette classe **autant de fois que l'on souhaite produire d'objets**
- Métaphore du moule pour les objets manufacturés

<b>avec moulage</b>	chaque objet est obtenu à partir d'une empreinte commune
<b>sans moulage</b>	chaque objet doit être construit de A à Z

# Communication inter-objets

Un programme développé avec le paradigme de la POO est **constitué d'objets** ! Ces derniers communiquent à l'aide de "*messages*".

L'interaction entre objets peut être représentée par le concept abstrait d'envoi de messages, on dit que des objets communiquent lorsqu'ils échangent un message :

- ce message de l'objet émetteur vers l'objet récepteur correspond à une demande de traitement
- selon les traitements qu'il est capable de réaliser, le récepteur réagit et peut même répondre à l'émetteur

# Communication par messages

## Communication

Les objets **interagissent** et **communiquent** entre eux par l'envoi de "messages"

- Les méthodes publiques d'un objet correspondent aux messages qu'il peut recevoir
- Un message est composé :

# Communication par messages

## Communication

Les objets **interagissent** et **communiquent** entre eux par l'envoi de "messages"

- Les méthodes publiques d'un objet correspondent aux messages qu'il peut recevoir
- Un message est composé :
  - ▶ du nom de l'objet cible (récepteur) du message

# Communication par messages

## Communication

Les objets **interagissent** et **communiquent** entre eux par l'envoi de "messages"

- Les méthodes publiques d'un objet correspondent aux messages qu'il peut recevoir
- Un message est composé :
  - ▶ du nom de l'objet cible (récepteur) du message
  - ▶ du nom de la méthode à exécuter



# Communication par messages

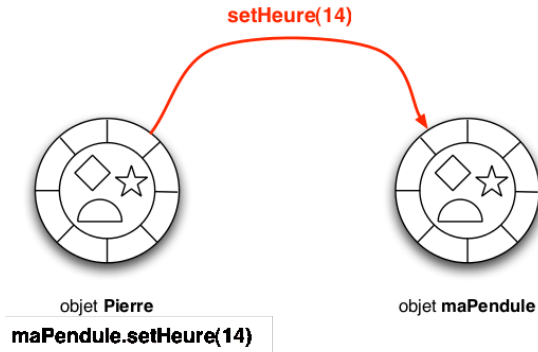
## Communication

Les objets **interagissent** et **communiquent** entre eux par l'envoi de "messages"

- Les méthodes publiques d'un objet correspondent aux messages qu'il peut recevoir
- Un message est composé :
  - ▶ du nom de l'objet cible (récepteur) du message
  - ▶ du nom de la méthode à exécuter
  - ▶ des paramètres nécessaires à cette méthode

# Échange de messages

Les objets s'envoient des messages entre eux



# Cycle de vie d'un objet

## Cycle de vie

- le cycle de vie d'un objet correspond à :
  - ① sa naissance : **création** de l'objet (appel d'une méthode particulière appelée le **constructeur**)
  - ② sa vie : l'objet réagit aux messages qui lui sont adressés et réalise des traitements en conséquence
  - ③ sa mort : l'objet est définitivement détruit
- En Java, nous verrons que le langage gère la destruction des objets pour nous !

Tout objet doit être créé avant de pouvoir effectuer des traitements  
!!!

On doit donc en premier lieu appeler son **constructeur** !

## Récapitulatif classes, objets, attributs et méthodes en Java

Une classe définit un **nouveau type de données** caractérisé par :

- **attributs** (données spécifiques à cette classe)
- **méthodes** (comportement des objets de cette classe)
- droits d'accès :
  - ▶ certaines méthodes et attributs sont **“cachés”** en utilisant le mot clé **private** : état interne d'un objet et détails d'implémentation
  - ▶ d'autres constituent l'“interface” de cette classe et sont déclarés avec le mot clé **public**
  - ▶ il existe d'autres modes de protection que l'on verra plus tard (cf. CM sur l'héritage)
- une instanciation particulière d'une classe s'appelle un **objet** (une **instance**)

# Mini Quiz



Votons à mains levées !

## Une instance (ou objet) ...

- 1 est créée à partir d'une classe.
- 2 permet de créer de multiples classes.
- 3 possède au maximum un attribut de type non primitif.
- 4 est forcément privée.

## Une méthode ...

- 1 est l'“équivalent ” (à peu près) d'une fonction en C.
- 2 doit forcément être déclarée à l'intérieur d'une classe.
- 3 est obligatoirement appelée à partir d'un objet.

## Une classe ...

- 1 ne permet pas la création de plusieurs objets.
- 2 regroupe l'ensemble de l'état et des comportements d'une famille d'objets.
- 3 est forcément privée.
- 4 peut être écrite par vous.

# Ma première classe Java I

Une classe *PersonnageEx* 

```
/**
 * Classe exemple d'un personnage basique ayant seulement
 * un nombre de points de vie et un nombre de points d'action (energie)
 * @author jmnormand
 * @version 1.0
 */
public class PersonnageEx {
    // Attributs de la classe
    /**
     * Quantite d'energie du personnage (prive)
     */
    private int energie;
```

# Ma première classe Java II

```
/**
 * Nombre maximum de points de vie du personnage (prive)
 */
private int ptVie;
// Constructeurs et methodes
/**
 * Un constructeur avec deux entiers
 * en parametres.
 * @param e un entier representant l'energie du personnage
 * @param pv un entier representant les points de vie du personnage
 */
public PersonnageEx(int e, int pv) {
    energie = e;
    ptVie = pv;
}
```

# Ma première classe Java III

```
/**
 * Methode publique affichant le type de l'objet
 */
public void afficheType() {
    System.out.println("Objet de la classe PersonnageEx");
}
} // Fin de la classe
```



## Droits d'accès **privé** et **protégé**

### Mot-clé **private**

Tout ce qui relève des détails d'implémentation constitue l'état (interne) de l'objet et est identifié par le mot clé **private** :

- attribut et/ou méthode **privé(e)** = inaccessible depuis l'extérieur de la classe
- erreur de compilation si un programmeur-utilisateur essaye d'accéder à un(e) attribut/méthode privé(e)

NB : il existe également un mot clé **protected** qui permet de restreindre l'accès à des attributs et méthodes  $\Rightarrow$  nous y reviendrons plus tard

## Droits d'accès **public** et droit d'accès **par défaut**

### Mot-clé **public**

Tout ce qui est mis à disposition de l'extérieur (i.e. au programmeur-utilisateur) se déclare avec le mot-clé **public**

### Droit d'accès par défaut

#### Si aucun droit d'accès spécifié

Il s'agit des droits d'accès par défaut ! (**friendly**).

Si aucun droit d'accès n'est spécifié alors l'attribut ou la méthode est publiquement accessible par toutes les autres classes du même **paquetage** (**package**), mais pas en dehors de celui-ci !

⇒ **Lire absolument le document sur les paquetages sur le serveur pédagogique !**

## Récapitulatif sur les droits d'accès

### Dans la plupart des cas :

- On protège (mot-clé `private`) :
  - ▶ **tous les attributs**
  - ▶ des méthodes "internes"
- On rend accessible (mot-clé `public`):
  - ▶ des méthodes bien choisies (l'interface pour le programmeur-utilisateur)

### Recommandations

Mettre explicitement `public` ou `private` devant tous les attributs et toutes les méthodes de vos classes ! Ne jamais laisser un attribut ou une méthode sans droit d'accès (i.e. droit d'accès par défaut) !

# Déclaration de méthodes (1)

La syntaxe de la définition des méthodes en Java ressemble à celle vue en ALGPR pour les fonctions C/C++ (en ajoutant le droit d'accès) :

```
droitAcces typeRetour nomMethode(typeParam1 nomParam1, ...)  
{  
    // corps de la méthode  
}
```

mais elles sont **déclarées et définies dans la classe elle même** !

## Déclaration de méthodes (2)

Exemple : la méthode `afficheType` de la classe `PersonnageEx`

```
public void afficheType() {  
    System.out.println("Objet de la classe PersonnageEx");  
}
```

Les méthodes sont :

- des fonctions/procédures propres à la classe
- **qui ont accès directement aux attributs de la classe sans avoir à les passer en paramètres**
- mais qui peuvent avoir besoin de paramètres autres que les attributs

# Appel de méthodes

L'appel à une méthode `nomMethode` à partir d'une instance nommée `nomInstance` s'écrit de la manière suivante :

```
nomInstance.nomMethode(valeurParam1, ...);
```

## Exemple : appel de méthode

Appel de la méthode `afficheType` par l'instance `perso` de la classe `PersonnageEx` :

```
perso.afficheType();
```

## Surcharge de méthodes (1)

**En Java, il est possible de définir plusieurs fois la même méthode !**

Supposons que l'on souhaite manipuler une date dans un programme. On peut imaginer vouloir changer la date :

Changer la date en passant jour, mois et année

```
int changerDate(int jour, int mois, int annee){  
    ...  
}
```

OK mais si on veut simplement changer l'année de notre date :  $\Rightarrow$   
comment faire si l'on ne veut pas modifier ni le jour, ni le mois ?  
Solutions envisageables :

## Surcharge de méthodes (1)

**En Java, il est possible de définir plusieurs fois la même méthode !**

Supposons que l'on souhaite manipuler une date dans un programme. On peut imaginer vouloir changer la date :

Changer la date en passant jour, mois et année

```
int changerDate(int jour, int mois, int annee){  
    ...  
}
```

OK mais si on veut simplement changer l'année de notre date :  $\Rightarrow$  comment faire si l'on ne veut pas modifier ni le jour, ni le mois ?

Solutions envisageables :

- passer "-1" dans les paramètres inutilisés  $\Rightarrow$  pas très logique !



## Surcharge de méthodes (1)

En Java, il est possible de définir plusieurs fois la même méthode !

Supposons que l'on souhaite manipuler une date dans un programme. On peut imaginer vouloir changer la date :

Changer la date en passant jour, mois et année

```
int changerDate(int jour, int mois, int annee){  
    ...  
}
```

OK mais si on veut simplement changer l'année de notre date :  $\Rightarrow$  comment faire si l'on ne veut pas modifier ni le jour, ni le mois ?

Solutions envisageables :

- passer "-1" dans les paramètres inutilisés  $\Rightarrow$  pas très logique !
- déclarer d'autres méthodes :

```
int changerAnneeDate(int jour){ /* corps de la methode */ }
```

$\Rightarrow$  OK mais on va avoir potentiellement beaucoup de méthodes !

## Surcharge de méthodes (1)

En Java, il est possible de définir plusieurs fois la même méthode !

Supposons que l'on souhaite manipuler une date dans un programme. On peut imaginer vouloir changer la date :

Changer la date en passant jour, mois et année

```
int changerDate(int jour, int mois, int annee){  
    ...  
}
```

OK mais si on veut simplement changer l'année de notre date :  $\Rightarrow$  comment faire si l'on ne veut pas modifier ni le jour, ni le mois ?

Solutions envisageables :

- passer "-1" dans les paramètres inutilisés  $\Rightarrow$  pas très logique !
- déclarer d'autres méthodes :  

```
int changerAnneeDate(int jour){ /* corps de la methode */ }
```

 $\Rightarrow$  OK mais on va avoir potentiellement beaucoup de méthodes !
- quoi d'autre ?

## Surcharge de méthodes (2)

### La solution en Java : surcharge de méthode

En Java il est possible de définir plusieurs fois une méthode ayant le même nom, en **respectant un certain nombre de règles !**

On peut donc écrire :

```
// surcharge de methodes
int changerDate(int jour, int mois, int annee){
    ...
}
int changerDate(int mois, int annee){
    ...
}
int changerDate(int annee){
    ...
}
```

## Surcharge de méthodes (3)

Il est tout à fait possible de définir plusieurs fois la même méthode si et seulement si :

- le **nombre** des paramètres est différent
- le **type** des paramètres est différent **si leur nombre est identique**.

Ce mécanisme s'appelle la **surcharge de méthode**, appelé **overload** en anglais !

### Comment **NE PAS** surcharger une méthode !

Attention, le mécanisme de surcharge ne s'applique pas dans l'un ou l'autre des cas suivants (qui aboutissent à une erreur de compilation) :

- changer **uniquement** le type de retour de la méthode
- changer **uniquement** le nom des paramètres de la méthode (et pas leurs types)

## Surcharge de méthodes IV

### Exemples **valides** de surcharge de méthode

```
// surcharge OK : le nombre de parametres est different
int changerDate(int annee, int mois){
```

```
    ...
}
```

```
int changerDate(int annee){
```

```
    ...
}
```

```
// surcharge OK : les types des parametres sont differents
```

```
int changerDate(float annee){
```

```
    ...
}
```

```
int changerDate(int annee){
```

```
    ...
}
```

## Surcharge de méthodes IV

### Exemples **invalides** de surcharge de méthode

```
// surcharge pas OK : erreur de compilation ! Seul le type de retour
// change (1 methode retourne 1 entier l'autre 1 flottant)
int changerDate(int annee){
    ...
}
float changerDate (int annee){
    ...
}
// surcharge pas OK : erreur de compilation !
// seul le nom du parametre est different (ici de 'annee' a 'mois')
int changerDate(int annee){
    ...
}
int changerDate(int mois){
    ...
}
```

## Accesseurs (Getters) et Modificateurs (Setters) aux attributs d'une classe

### Modification/Accès aux attributs privés

Si les attributs ne sont pas publics, comment y accède-t-on, comment les modifie-t-on, depuis l'extérieur de la classe ?

On inclut les méthodes publiques nécessaires :

- ➊ **Accesseurs** : permettent d'accéder (en lecture seule) aux valeurs des attributs en retournant la valeur d'un attribut
- ➋ **Modificateurs** : permettent de modifier (en écriture) les valeurs des attributs en affectant aux attributs une nouvelle valeur

# Accesseurs, modificateurs et encapsulation

Pourquoi écrire des accesseurs/modificateurs alors que l'on peut tout mettre en **public** ?

- pour garantir la cohérence des objets !



# Accesseurs, modificateurs et encapsulation

Pourquoi écrire des accesseurs/modificateurs alors que l'on peut tout mettre en **public** ?

- pour garantir la cohérence des objets !

## Classe PersonnageEx avec attributs publiques

```
public class PersonnageEx {  
    public int energie;  
    public int ptVie;  
    public boolean estEnVie;  
    // autres attributs, methodes etc.  
    public void setPointsVie(int pv) {  
        ptVie = pv;  
        if(ptVie < 0) {  
            estEnVie = false;  
        }  
    }  
}
```

## Accesseurs, modificateurs et encapsulation

Pourquoi écrire des accesseurs/modificateurs alors que l'on peut tout mettre en **public** ?

- pour garantir la cohérence des objets !

Problème d'accès à un attribut public

```
// quelque part dans un programme principal  
p.ptVie = -100;  
System.out.println(p.ptVie);  
System.out.println(p.estEnVie);
```

Affichage du résultat

```
-100  
true
```

## Accesseurs, modificateurs et encapsulation

Pourquoi écrire des accesseurs/modificateurs alors que l'on peut tout mettre en **public** ?

- pour garantir la cohérence des objets !

Problème d'accès à un attribut public

```
// quelque part dans un programme principal
p.setPointsVie(-100);
System.out.println(p.ptVie);
System.out.println(p.estEnVie);
```

Affichage du résultat

```
-100
false
```

# Classe PersonnageEx mise à jour I

Une classe `PersonnageEx` 

```
/**
 * Classe exemple d'un personnage basique ayant seulement
 * un nombre de points de vie et un nombre de points d'action (energie)
 * @author jmnormand
 * @version 1.0
 */
public class PersonnageEx {
    // Attributs de la classe
    /**
     * Quantite d'energie du personnage (prive)
     */
    private int energie;

    /**
```

## Classe PersonnageEx mise à jour II

```
* Nombre maximum de points de vie du personnage (prive)
*/
private int ptVie;

// Constructeurs et methodes
/**
 * Un constructeur prenant deux entiers en parametres.
 * @param e un entier representant l'energie du personnage
 * @param pv un entier representant les points de vie du personnage
 */
public PersonnageEx(int e, int pv) {
    energie = e;
    ptVie = pv;
}
```

## Classe PersonnageEx mise à jour III

```
/**
 * Methode publique affichant le type de l'objet
 */
public void afficheType() {
    System.out.println("Classe PersonnageEx");
}

// Accesseurs et modificateurs

/**
 * Accesseur sur l'attribut 'energie'
 */
public int getEnergie() {
    return energie;
}
```

## Classe PersonnageEx mise à jour IV

```
/**
 * Accesseur sur l'attribut 'ptVie'
 */
public int getPointsVie() {
    return ptVie;
}

/**
 * Modificateur de l'attribut 'energie'
 */
public void setEnergie(int e) {
    energie = e;
}
```

## Classe PersonnageEx mise à jour V

```
/**
 * Modificateur de l'attribut 'ptVie'
 */
public void setPointsVie(int pv) {
    ptVie = pv;
}

// Fin de la classe
}
```



## Mini Quiz

Votons à mains levées !

Quelle(s) affirmation(s) est/sont correcte(s) ?

- ❶ Il est obligatoire de mettre un droit d'accès dans l'entête de déclaration d'une méthode.
- ❷ Les méthodes n'ont pas d'accès direct aux attributs de la classe.
- ❸ On peut trouver plusieurs méthodes de même nom dans une (même) classe.
- ❹ Les accesseurs et mutateurs sont obligatoires dans une classe Java.

Soient un objet `o` et une méthode `f` d'une classe `A`

On écrit :

- ❶ `A.f(o)`
- ❷ `A.o.f(...)`
- ❸ `o.f(...)`
- ❹ `f.o(...)`

## Masquage (Shadowing) de variables/attributs

Attention, en Java rien ne vous interdit d'écrire ceci :

### Exemple de masquage

```
public void setEnergie(int energie) {  
    energie = energie; // what??!!??  
}
```

que signifie cette instruction ?

C'est une situation classique en POO :

- le paramètre d'une méthode "masque" un attribut de la classe
- masquage = un identificateur "masque/cache" un autre identificateur

## Masquage et mot clé `this`

Il est possible d'expliciter que l'on veut accéder à un attribut d'une classe, et ainsi éviter le phénomène de **masquage**, en utilisant le mot clé `this` :

- `this` est un **“raccourci” vers l'instance/objet courant(e)**
- `this`  $\simeq$  “moi”

Syntaxe d'accès à un attribut en cas de masquage (ou d'ambiguïté)

```
this.nomAttribut
```

Résolution du problème de masquage avec le mot clé `this`

```
public void setEnergie(int energie) {  
    this.energie = energie; // c'est plus comprehensible maintenant  
}
```

L'utilisation de `this` est obligatoire en situation de **masquage**  $\Rightarrow$  vous devez éviter ces situations !

## Mot clé `this`

- Plus généralement, quand à l'intérieur d'une classe on accède à des **attributs** ou des **méthodes**, on fait implicitement référence à ceux de l'**objet courant**
- On recommande donc de toujours utiliser le mot clé `this` à l'intérieur d'une classe pour éviter les ambiguïtés (masquage)
- `this` est aussi utilisé lorsque l'objet courant se passe lui même comme paramètre à une autre méthode

### Résolution du problème de masquage avec le mot clé `this`

```
public class A {  
    // ...  
    public void setB(B b){  
        // ...  
    }  
}
```

```
public class B {  
    // ...  
    public void someMethod(A myObjA) {  
        myObjA.setB(this);  
    }  
}
```

# Comment construire des objets ?

Jusqu'ici, nous avons vu :

- comment écrire une classe
- comment définir, avoir accès et modifier des attributs
- comment déclarer et appeler des méthodes
- manipuler des objets

Mais il nous manque la 1<sup>re</sup> étape du cycle de vie des objets (cf. slide 65) !

Nous ne sommes pas encore capables de **créer un objet** !  $\Rightarrow$  ce processus s'effectue grâce à **l'appel à un constructeur** en Java

# Constructeur

Un constructeur est une méthode “presque” comme les autres :

- pas de type de retour (pas même `void`)
- **même nom** que la classe
- doit être invoquée **SYSTÉMATIQUEMENT** à chaque fois qu’une instance est créée !!
- peut être surchargé i.e. possibilité pour une classe d’avoir plusieurs constructeurs, si **leurs listes de paramètres sont différentes**

## But du constructeur

Un constructeur doit créer un objet et son but est **d’initialiser tous les attributs** de l’objet nouvellement créé !

## Déclaration d'un constructeur

La syntaxe de base d'un constructeur en Java est la suivante :

```
droitAcces NomClasse(liste_paramètres)
{
    // initialisation des attributs
    // en utilisant liste_paramètres
}
```

### Exemple : un constructeur de la classe `PersonnageEx`

```
// Constructeur de notre classe PersonnageEx prenant en
// parametres 2 entiers qui vont nous permettre d'initialiser
// les valeurs de nos deux attributs 'energie' et 'ptVie'
public PersonnageEx(int e, int pv) {
    this.energie = e; // equivalent a : energie = e;
    this.ptVie = pv; // equivalent a : ptVie = pv;
}
```

## Création d'un objet - Appel d'un constructeur

La déclaration avec initialisation d'un objet se fait par la syntaxe suivante :

```
NomClasse nomInstance = new NomClasse(valArg1, ..., valArgN);
```

L'appel au mot-clé `new` et à un constructeur permet de créer un **nouvel** objet :

- utilisation du mot clé Java `new` : qui crée le nouvel objet en mémoire
- suivi de l'appel au constructeur : qui initialise l'objet (i.e. ses attributs) grâce aux valeurs des arguments passés en paramètres (`valArg1, ..., valArgN`)



## Création d'un objet - Appel d'un constructeur

L'appel au mot-clé `new` et à un constructeur permet de créer un **nouvel objet** :

- utilisation du mot clé Java `new` : qui crée le nouvel objet en mémoire
- suivi de l'appel au constructeur : qui initialise l'objet (i.e. ses attributs) grâce aux valeurs des arguments passés en paramètres (`valArg1`, ..., `valArgN`)

### Appel de constructeurs

```
// Appel au constructeur par défaut
PersonnageEx p1 = new PersonnageEx();

// Appel au constructeur avec un parametre
PersonnageEx p2 = new PersonnageEx(15);

// Appel au constructeur avec deux parametres
PersonnageEx p3 = new PersonnageEx(50,26);
```

## Constructeur par défaut

Le constructeur par défaut est celui qui n'**a pas de paramètre** :

- il attribue donc aux attributs des valeurs par défaut

### Exemples de constructeurs

```
// Constructeur par défaut
```

```
public PersonnageEx() {  
    energie = 100;  
    ptVie = 100;  
}
```

```
// Constructeur avec 2 parametres
```

```
public PersonnageEx(int e, int pv) {  
    energie = e;  
    ptVie = pv;  
}
```

```
// Constructeur avec 1  
    parametre
```

```
public PersonnageEx(int e) {  
    energie = e;  
    ptVie = e;  
}
```

## Impact de la manière de construire un objet sur notre vision de la représentation des variables, objets, etc. en Java

Jusqu'ici (en ALGPR et depuis le début de cette séance), nous avons vu que pour utiliser des variables de type simple (ou type de base, i.e. `float`, `int`, etc.), il nous suffisait d'écrire des instructions semblables à :

### Manipulation de variables de type de base

```
// dans une methode ou dans un main
int i = 3; // declaration + initialisation de la variable 'i'
int j = 4; // declaration + initialisation de la variable 'j'
int k; // declaration de la variable 'k'
    // (NB : Java lui affectera automatiquement une valeur
    // par défaut, ici '0' puisque 'k' est un entier)
k = i + j; // initialisation de la variable 'k'
System.out.println(k); // affichage de la variable 'k'
```

## Impact de la manière de construire un objet sur notre vision de la représentation des variables, objets, etc. en Java

Jusqu'ici (en ALGPR et depuis le début de cette séance), nous avons vu que pour utiliser des variables de type simple (ou type de base, i.e. `float`, `int`, etc.), il nous suffisait d'écrire des instructions semblables à :

Créer des objets via l'appel à un constructeur change la donne

Pour les objets, il faut utiliser `new` et l'appel à un constructeur :

```
// dans une methode ou dans un main
```

```
PersonnageEx p = new PersonnageEx(10,50); // declaration et  
        initialisation de l'objet 'p'
```

```
PersonnageEx p2; //declaration de 'p2' (NB : Java lui attribue  
// aussi une valeur par default, null, spécifique aux objets)
```

```
p2=new PersonnageEx(50,120);// init. (= creation) de l'objet 'p2'  
// par appel a new et au constructeur de la classe
```

## Impact de la manière de construire un objet sur notre vision de la représentation des variables, objets, etc. en Java

Jusqu'ici (en ALGPR et depuis le début de cette séance), nous avons vu que pour utiliser des variables de type simple (ou type de base, i.e. `float`, `int`, etc.), il nous suffisait d'écrire des instructions semblables à :

### Représentation mémoire en Java

Cette différence n'est pas si anodine et s'explique par la manière **différente** dont Java stocke les variables de type de base et les objets en mémoire !

## Mini Quiz

Votons à mains levées !

### Le constructeur d'une classe ...

- 1 doit absolument être invoqué pour créer un objet **utilisable**.
- 2 s'écrit presque comme une méthode classique.
- 3 a pour but d'initialiser les valeurs de tous les attributs d'une classe.
- 4 n'a pas besoin du mot clé **new** pour fonctionner.

### Il est possible ...

- 1 d'écrire plusieurs constructeurs dans la même classe.
- 2 de ne pas écrire un seul constructeur dans une classe.
- 3 de créer un objet sans appel au constructeur.

### Soient la classe **MaC** et un objet **monObj**

...

Comment écrit t'on la déclaration et l'initialisation de **monObj** ?

- 1 **monObj() = MaC;**
- 2 **MaC monObj() = new MaC;**
- 3 **monObj(MaC);**
- 4 **MaC monObj = new MaC();**

## Variables, objets, références en Java (1)

En Java les données de types de base, les données de types composés (tableaux) et les objets **NE SONT PAS STOCKÉS EN MÉMOIRE DE LA MÊME MANIÈRE !**

# Variables, objets, références en Java (1)

En Java les données de types de base, les données de types composés (tableaux) et les objets **NE SONT PAS STOCKÉS EN MÉMOIRE DE LA MÊME MANIÈRE !**

## Variables de type de base

Toute variable de **type de base** stocke directement une VALEUR

### Variable de type simple

```
// declaration d'une variable  
// de type simple 'double'  
double pi = 3.14159;
```

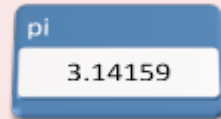


Figure: Représentation d'une variable de type simple.



# Variables, objets, références en Java (1)

En Java les données de types de base, les données de types composés (tableaux) et les objets **NE SONT PAS STOCKÉS EN MÉMOIRE DE LA MÊME MANIÈRE !**

## Variables de type évolué

Toute variable de **type de évolué** (tableaux, chaînes de caractères, objets) stocke une **référence** (adresse) vers une valeur

### Variable de type évolué

```
// déclaration d'une variable  
// de type composé 'String'  
String s = "Hello";
```

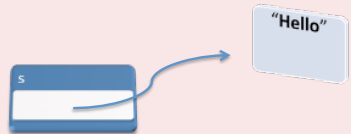


Figure: Représentation d'une variable d'un type composé/objet.

## Variables, objets, références en Java (2)

### Objet

Un objet est représenté en mémoire via une **référence** (adresse) vers un espace mémoire stockant l'objet :

#### Exemple création objet

```
// Appel constructeur avec 2 param.  
PersonnageEx p=new  
    PersonnageEx(50,26);
```

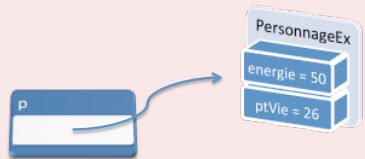


Figure: Représentation d'un objet en mémoire.

# Implication sur l'affectation en Java

Cette représentation sous forme de référence pour les variables de type évolués (tableaux, objets, etc.) a une très grande incidence sur la sémantique de l'**affectation** en Java (opérateur =):

## Affectation en Java

```
// affectation de v2 a v1  
v1 = v2;
```

## Que se passe t'il ?

- modification de la valeur stockée dans v1 ?
- modification de l'objet référencé par v1 ?



## Implication sur la comparaison en Java

Cela a également une très grande incidence sur la sémantique de la **comparaison** en Java (opérateur `==`) :

### Comparaison en Java

```
// comparaison entre v2  
    et v1  
if(v1 == v2) ...
```

### Que se passe t'il ?

- comparaison des références ?
- comparaison des valeurs ?

# Implication sur l'affichage en Java

Mais aussi sur la sémantique de l'**affichage** en Java (opération `System.out.println`) :

## Affichage en Java

```
// affichage de v1  
System.out.println(v1);
```

## Que se passe t'il ?

- affichage de la valeur de `v1` ?
- affichage de la référence contenue dans `v1` ?
- etc.

## Le mot clé `null` en Java

Le mot clé `null` en Java représente la **référence nulle**, c'est à dire **l'absence d'objet** :

- peut être affecté à n'importe quel objet de n'importe quelle classe
- affecté à une variable, cela indique que la variable ne référence **aucun objet**
- représente l'**objet "vide"** et est une zone réservée dans la mémoire
- il peut être judicieux, avant d'utiliser un objet, de vérifier que celui-ci n'est pas nul

### Classe `PersonnageEx` v1

```
PersonnageEx p1 = null; // p1 ne reference aucun objet
PersonnageEx p2; // p2 non plus
// ...
if(p1 != null){...}
if(p2 == null){...} // Dans quel 'if' va t'on passer ????
```

# Le mot clé `null` et les objets

Illustration de création d'objets et mot clé `null` :

## Le mot clé `null` et les objets

Illustration de création d'objets et mot clé `null` :

### Exemple création objet

```
// Appel au constructeur  
// avec deux parametres  
PersonnageEx p;
```



Figure: Représentation d'un objet en mémoire.



## Le mot clé `null` et les objets

Illustration de création d'objets et mot clé `null` :

### Exemple création objet

```
// Appel au constructeur  
// avec deux parametres  
PersonnageEx p;  
p = new PersonnageEx(50,26);
```

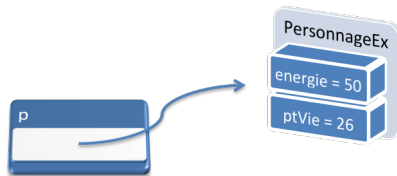


Figure: Représentation d'un objet en mémoire.

## Le mot clé `null` et les objets

Illustration de création d'objets et mot clé `null` :

### Exemple création objet

```
// Appel au constructeur  
// avec deux parametres  
PersonnageEx p;  
p = new PersonnageEx(50,26);  
PersonnageEx p2;
```

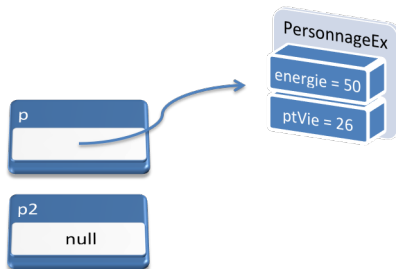


Figure: Représentation d'un objet "vide" en mémoire.

## Le mot clé `null` et les objets

Illustration de création d'objets et mot clé `null` :

### Exemple création objet

```
// Appel au constructeur  
// avec deux parametres  
PersonnageEx p;  
p = new PersonnageEx(50,26);  
PersonnageEx p2;  
p2 = p;
```

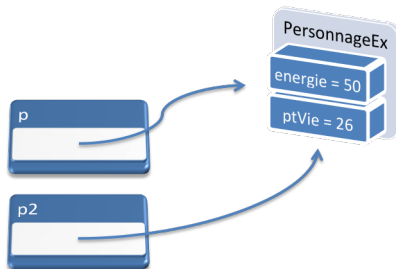


Figure: Objets et références.

# Constructeur par défaut “par défaut” de Java (1)

## Constructeur par défaut

Rappel : on appelle **constructeur par défaut** un constructeur qui ne prend pas de paramètre.

## Le constructeur par défaut de `PersonnageEx`

```
// Constructeur par default
public PersonnageEx() {
    energie = 100;
    ptVie = 100;
}
```

## Constructeur par défaut “par défaut” de Java (2)

- Si aucun constructeur n'est spécifié, le compilateur Java **génère automatiquement** pour vous une **version minimale du constructeur par défaut** qui initialise les attributs :
  - ▶ de types objets à **null**
  - ▶ des types de base à leurs valeurs nulles (p. ex. 0 pour les entiers)
- Dès qu'**au moins un constructeur est spécifié** dans une classe, ce constructeur “par défaut par défaut” n'est **plus fourni** par Java !
- Si l'on spécifie **une** classe sans constructeur par défaut  $\Rightarrow$  il n'est plus possible d'instancier cette classe sans donner de paramètres au constructeur!

Supposons une classe **PersonnageBis** identique à **PersonnageEx** mais sans constructeur par défaut

```
// PersonnageBis n'a pas de constructeur par default
PersonnageBis pb;
pb = new PersonnageBis(); // OK ? valeur des attributs ?
```

# Création d'un objet : exemples

## Classe PersonnageEx v1

```
public class
    PersonnageEx {
    private int
        energie;
    private int ptVie;
    // pas de
    // constructeur
    // ...
}
```

## Classe PersonnageEx v2

```
public class
    PersonnageEx {
    private int
        energie;
    private int ptVie;

    public
        PersonnageEx() {
        energie = 100;
        ptVie = 100;
        }

    // pas d'autre
    // constructeur
    // ...
}
```

## Classe PersonnageEx v3

```
public class
    PersonnageEx {
    private int energie;
    private int ptVie;

    public
        PersonnageEx(int e,
            int pv) {
        energie = e;
        ptVie = pv;
        }


    // pas d'autre
    // constructeur
    // ...
}
```

## Création d'un objet : exemples

Version de la classe	PersonnageEx p1; p1 = <b>new</b> PersonnageEx();	PersonnageEx p2; p2 = <b>new</b> PersonnageEx(50,26);
v1	?	?
v2	?	?
v3	?	?

**Table:** Résultats d'appels à des constructeurs en fonction de la définition de la classe PersonnageEx.

## Création d'un objet : solution

Version de la classe	PersonnageEx p1; p1 = <b>new</b> PersonnageEx();	PersonnageEx p2; p2 = <b>new</b> PersonnageEx(50,26);
v1	<i>energie</i> = 0 <i>ptVie</i> = 0 	Erreur !
v2	<i>energie</i> = 100 <i>ptVie</i> = 100	Erreur !
v3	Erreur !	<i>energie</i> = 50 <i>ptVie</i> = 26

**Table:** Résultats d'appels à des constructeurs en fonction de la définition de la classe PersonnageEx.



# Constructeur de recopie en Java (1)

Java offre un moyen de créer la **copie** d'une instance en utilisant un **constructeur de recopie** (aussi appelé **constructeur de copie**)

## Utilisation d'un constructeur de recopie en Java

```
PersonnageEx p1 = new PersonnageEx(75,58);  
PersonnageEx p2 = new PersonnageEx(p1);
```

p1 et p2 sont deux instances **distinctes** mais ayant **initialement** les mêmes valeurs pour leurs attributs

## Constructeur de copie en Java (2)

Le constructeur de copie permet d'initialiser une nouvelle instance en copiant les valeurs des attributs d'une autre instance du même type

### Déclaration d'un constructeur de copie en Java

```
public PersonnageEx(PersonnageEx p)
{
    energie = p.energie; // ou energie = p.getEnergie();
    ptVie = p.ptVie; // ou ptVie = p.getPointsVie();
}
```

### Attention car en Java :

- il n'y a pas de constructeur de copie fourni par défaut !
- il est possible de créer une copie d'objet en utilisant la méthode `clone()` (qu'il vous faudra redéfinir, attention !)

## Fin de vie d'un objet

Un objet est en fin de vie lorsque le programme n'en a plus besoin  $\Rightarrow$  la référence qui lui est associée n'est plus utilisée nulle part

### Fin de vie

```
public class FinDeVie {  
    public static void main(String[] args)  
    {  
        // des trucs Java  
        afficheEnergiePerso(10,20);  
        // encore des trucs ...  
    }  
}
```

### Fin de vie

```
public static void  
    afficheEnergiePerso(int e, int p)  
{  
    PersonnageEx perso = new  
        PersonnageEx(e,p);  
    System.out.println("le perso a  
        "+p+" points d'énergie");  
}  
// Fin de la classe FinDeVie
```

## Garbage Collection

Fin de vie = **récupération** de l'espace mémoire occupé par un objet  $\Rightarrow$  processus de *Garbage Collection* de Java

# Gestion de la mémoire en Java

- L'instanciation provoque une allocation dynamique d'espace mémoire sur chaque objet
- La machine virtuelle Java (JVM) maintient un compteur de références
- Lorsque le compteur atteint 0 (l'objet n'est plus référencé), la mémoire qui lui était allouée est **automatiquement "libérée"**
- En réalité ce n'est pas si simple :
  - ▶ un processus asynchrone, le *garbage collector* est chargé de récupérer cette mémoire libérée
  - ▶ pas de garantie sur l'instant de libération (sauf appel explicite)
  - ▶ très grosse différence avec C++ (opérateur `delete`)
  - ▶ plusieurs "types" de GC

## Garbage Collector - “Ramasse miettes” Java

- **Garbage Collector** = “Ramasse miettes” = processus léger (**thread**) qui s'exécute en tâche de fond avec une priorité faible :
  - ▶ lorsqu'il n'y a aucune activité
  - ▶ lorsque la JVM n'a plus de mémoire disponible (ralentit le système)
- moins efficace que la gestion explicite de la mémoire mais plus simple et plus sûr !
- possibilité de faire un appel explicite au Garbage Collector si on le souhaite : **System.gc()** ;
- le programmeur Java **n'a pas à libérer explicitement la mémoire**, Java le fait pour vous (contrairement à d'autres langages de programmation, p. ex. C++)

## Gestion de la mémoire - Finalisation

### En plus du GC ?

- un objet peut contenir d'autres ressources à libérer que la mémoire :
  - ▶ descripteur de fichiers,
  - ▶ socket réseau,
  - ▶ contexte graphique pour une interface graphique utilisateur,
  - ▶ etc.
- on doit alors utiliser une méthode de "finalisation" de l'objet :
  - ▶ doit s'appeler `finalize()`
  - ▶ pas d'arguments, type de retour `void`
  - ▶ invoquée juste avant que le GC ne récupère l'espace mémoire
  - ▶  $\neq$  destructeur en C++ !!
- aucune garantie sur l'ordre d'exécution des méthodes de finalisation
- en général il **ne faut PAS** redéfinir `finalize` sauf si vous avez vraiment une bonne raison !

## Affectation & copie d'objets (copie superficielle/profonde)

Que se passe t'il dans l'exemple suivant :

### Exemple copie 1

```
// trucs ...
PersonnageEx p1 = new PersonnageEx(150,99);
PersonnageEx p2 = p1;
p1.setEnergie(200);
System.out.println(p1.getEnergie());
System.out.println(p2.getEnergie());
// encore des trucs ...
```

## Affectation & copie d'objets (copie superficielle/profonde)

Que se passe t'il dans l'exemple suivant :

### Exemple copie 1

```
// trucs ...  
PersonnageEx p1 = new PersonnageEx(150,99);  
PersonnageEx p2 = p1;  
p1.setEnergie(200);  
System.out.println(p1.getEnergie());  
System.out.println(p2.getEnergie());  
// encore des trucs ...
```

### Affichage résultat

200 200



## Affectation & copie d'objets (copie superficielle/profonde)

Que se passe t'il dans l'exemple suivant :

### Exemple copie 1

```
// trucs ...
PersonnageEx p1 = new PersonnageEx(150,99);
PersonnageEx p2 = p1;
p1.setEnergie(200);
System.out.println(p1.getEnergie());
System.out.println(p2.getEnergie());
// encore des trucs ...
```

### Affichage résultat

200 200

### C'est une copie superficielle

p1 et p2 référencent ici le **même objet** ! Toute modification de p1 va modifier p2 et inversement

## Affectation & copie d'objets (copie superficielle/profonde) (2)

Que se passe t'il dans l'exemple suivant :

### Exemple copie 2

```
// trucs ...  
PersonnageEx p1 = new PersonnageEx(150,99);  
PersonnageEx p2 = new PersonnageEx(p1);  
p1.setEnergie(200);  
System.out.println(p1.getEnergie());  
System.out.println(p2.getEnergie());  
// encore des trucs ...
```

## Affectation & copie d'objets (copie superficielle/profonde) (2)

Que se passe t'il dans l'exemple suivant :

### Exemple copie 2

```
// trucs ...  
PersonnageEx p1 = new PersonnageEx(150,99);  
PersonnageEx p2 = new PersonnageEx(p1);  
p1.setEnergie(200);  
System.out.println(p1.getEnergie());  
System.out.println(p2.getEnergie());  
// encore des trucs ...
```

### Affichage résultat

200 150

## Affectation & copie d'objets (copie superficielle/profonde) (2)

Que se passe t'il dans l'exemple suivant :

### Exemple copie 2

```
// trucs ...  
PersonnageEx p1 = new PersonnageEx(150,99);  
PersonnageEx p2 = new PersonnageEx(p1);  
p1.setEnergie(200);  
System.out.println(p1.getEnergie());  
System.out.println(p2.getEnergie());  
// encore des trucs ...
```

### Affichage résultat

200 150

### C'est une copie **profonde**

Uniquement car nous avons défini le constructeur de recopie ! p1 et p2 référencent maintenant deux objets différents !

## Comparaison d'objets (1)

Que se passe t'il dans l'exemple suivant :

### Exemple comparaison d'objets

```
// ... dans une methode main
PersonnageEx p1 = new PersonnageEx(150,99);
PersonnageEx p2 = new PersonnageEx(150,99);
if(p1 == p2) {
    System.out.println("Les personnages sont identiques");
} else {
    System.out.println("Ces personnages sont differents");
}
```

## Comparaison d'objets (1)

Que se passe t'il dans l'exemple suivant :

### Exemple comparaison d'objets

```
// ... dans une methode main
PersonnageEx p1 = new PersonnageEx(150,99);
PersonnageEx p2 = new PersonnageEx(150,99);
if(p1 == p2) {
    System.out.println("Les personnages sont identiques");
} else {
    System.out.println("Ces personnages sont differents");
}
```

### Affichage résultat

Ces personnages sont differents

## Comparaison d'objets (1)

Que se passe t'il dans l'exemple suivant :

### Exemple comparaison d'objets

```
// ... dans une methode main
PersonnageEx p1 = new PersonnageEx(150,99);
PersonnageEx p2 = new PersonnageEx(150,99);
if(p1 == p2) {
    System.out.println("Les personnages sont identiques");
} else {
    System.out.println("Ces personnages sont differents");
}
```

### Affichage résultat

Ces personnages sont differents

### Pourquoi ?

L'opérateur de comparaison (==) va regarder si les deux variables p1 et p2 référencent le même objet ! ⇒ **Ce n'est pas le cas ici !**

## Comparaison d'objets (2)

### Explication de code

```
// ... dans une methode main  
PersonnageEx p1 = new  
    PersonnageEx(150,99);
```

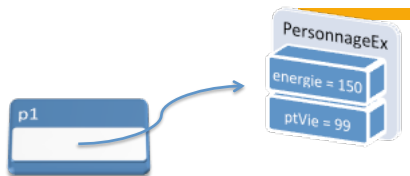


Figure: Création de p1.



## Comparaison d'objets (2)

### Explication de code

```
// ... dans une methode main  
PersonnageEx p1 = new  
    PersonnageEx(150,99);  
PersonnageEx p2 = new  
    PersonnageEx(150,99);
```

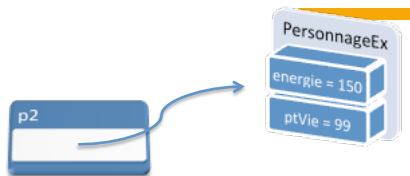


Figure: Création de p2.

## Comparaison d'objets (2)

### Explication de code

```
// ... dans une methode main
PersonnageEx p1 = new
    PersonnageEx(150,99);
PersonnageEx p2 = new
    PersonnageEx(150,99);
if(p1 == p2) {
    System.out.println("Les
        personnages sont identiques");
} else {
    System.out.println("Ces
        personnages sont differents");
}
```

### Affichage résultat

Ces personnages sont differents

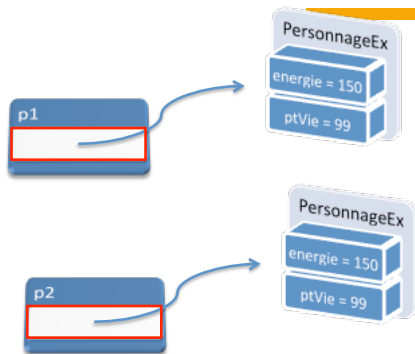


Figure: Comparaison de p1 avec p2.

## Comparaison d'objets (3)

Comment faire pour comparer deux objets ? (p. ex. de type `PersonnageEx`)

Il faut fournir une méthode qui compare les objets suivants des critères définis par le programmeur et qui ont du sens pour **le type** (i.e. la classe) des objets comparés.

Java prévoit pour ce faire plusieurs possibilités :

- la méthode `equals`, dont l'entête est **obligatoirement** le suivant (pour la classe `PersonnageEx`) :

```
public boolean equals(PersonnageEx p) { ... }
```

- la méthode `compareTo` (qui appartient à l'interface `Comparable` que nous verrons plus tard) et dont l'entête est le suivant (pour la classe `PersonnageEx`) :

```
public int compareTo(PersonnageEx p) { ... }
```

## Comparaison d'objets (4)

### Code pour la classe `PersonnageEx`

```
// fichier PersonnageEx.java
public class PersonnageEx {
    // voir declaration precedente de la classe PersonnageEx
    public boolean equals(PersonnageEx p) {
        if(p == null) {           Il faut tester ce n'est pas null
            return false;         d'abord
        } else {
            return ( (energie == p.getEnergie()) && (ptVie ==
p.getPointsVie()) );
        }
    }
}
```

## Comparaison d'objets (5)

Que se passe t'il dans l'exemple suivant :

Exemple comparaison d'objets avec `equals`

```
// ... dans une methode main
PersonnageEx p1 = new PersonnageEx(150,99);
PersonnageEx p2 = new PersonnageEx(150,99);
if(p1.equals(p2)) {
    System.out.println("Les personnages sont identiques");
}
else {
    System.out.println("Ces personnages sont differents");
}
```

## Comparaison d'objets (5)

Que se passe t'il dans l'exemple suivant :

Exemple comparaison d'objets avec `equals`

```
// ... dans une methode main
PersonnageEx p1 = new PersonnageEx(150,99);
PersonnageEx p2 = new PersonnageEx(150,99);
if(p1.equals(p2)) {
    System.out.println("Les personnages sont identiques");
}
else {
    System.out.println("Ces personnages sont differents");
}
```

Affichage résultat

Les personnages sont identiques

## Comparaison d'objets (6)

### Attention à l'entête de `equals`

Deux entêtes sont possibles pour la méthode `equals` :

- `boolean equals(UneClasse o)`
- `boolean equals(Object o)`

`Object` est une classe particulière de Java. Nous y reviendrons bientôt !

Les deux entêtes ne correspondent pas exactement à la même chose !  
L'une de ces deux méthodes implémente bien l'interface `Comparable` alors que l'autre non !

On en reparlera lorsque l'on aura vu ce que sont les `interfaces` en Java.

# Affichage d'objets (1)

Que se passe t'il dans l'exemple suivant :

## Exemple d'affichage d'objet

```
PersonnageEx p1 = new PersonnageEx(150,99);  
System.out.println(p1);
```



## Affichage d'objets (1)

Que se passe t'il dans l'exemple suivant :

### Exemple d'affichage d'objet

```
PersonnageEx p1 = new PersonnageEx(150,99);  
System.out.println(p1);
```

### Affichage résultat

PersonnageEx@1a06f956      Adresse d 'objet

## Affichage d'objets (1)

Que se passe t'il dans l'exemple suivant :

### Exemple d'affichage d'objet

```
PersonnageEx p1 = new PersonnageEx(150,99);  
System.out.println(p1);
```

### Affichage résultat

PersonnageEx@1a06f956

Cela affiche la valeur de la **référence** p1 !

C'est à dire le **nom** de la classe et l'**adresse mémoire** de l'objet référencé par **p1**

## Affichage d'objets (2)

Comment afficher les valeurs des attributs d'un objet en gardant le même code ?

- Java fournit une méthode spéciale qui permet d'obtenir une représentation de l'objet sous forme de chaîne de caractères (**String**).
- C'est la méthode **toString**, dont l'entête est **obligatoirement** :  
`public String toString() { ... }`
- La méthode **toString** est ensuite invoquée automatiquement par Java lors d'un appel à **System.out.println**
- il vous faudra donc **surcharger** cette méthode pour chacune de vos classes !

## Affichage d'objets (3)

### Exemple d'utilisation de la méthode `toString`

```
// fichier PersonnageEx.java
public class PersonnageEx {
    // voir declaration precedente de la classe PersonnageEx
    public String toString()
    {
        String res = "Personnage avec "+ptVie+" points de vie, et "+energie+"
            points d'energie";
        return res;
    }
}

// fichier TestPersEx.java
public class TestPersEx {
    public static void main(String[] args) {
        PersonnageEx p = new PersonnageEx(25,75);
        System.out.println(p);
    }
}
```

## Affichage d'objets (3)

### Exemple d'utilisation de la méthode `toString`

```
// fichier PersonnageEx.java
public class PersonnageEx {
    // voir declaration precedente de la classe PersonnageEx
    public String toString()
    {
        String res = "Personnage avec "+ptVie+" points de vie, et "+energie+"
            points d'energie";
        return res;
    }
}

// fichier TestPersEx.java
public class TestPersEx {
    public static void main(String[] args) {
        PersonnageEx p = new PersonnageEx(25,75);
        System.out.println(p);
    }
}
```

# Une parenthèse sur la modélisation

Nous avons vu jusqu'à présent :

- ce qu'est un **objet**
- ce qu'est une **classe**
- ce que sont des **attributs**
- ce que sont des **méthodes**
- ce que sont les **constructeurs**
- ce que sont les **droits d'accès** aux attributs et aux méthodes
- comment **écrire une classe** en Java
- comment **écrire un programme** (= fonction principale **main**) en Java
- comment **créer un objet**
- comment **appeler des méthodes d'un objet**
- comment **affecter, comparer, afficher** (textuellement) des objets

## Une parenthèse sur la modélisation

Nous avons vu jusqu'à présent :

- ce qu'est un **objet**
- ce qu'est une **classe**
- ce que sont des **attributs**

OK mais comment comprendre les interactions entre les objets ?

De quoi a t'on besoin :

- visualiser (graphiquement) nos classes
- abstraire nos classes et nos programmes
- partager nos classes simplement avec d'autres programmeurs

en un mot **modéliser** !

- comment **appeler des méthodes d'un objet**
- comment **affecter, comparer, afficher** (textuellement) des objets

# UML : Unified Modelling Language

Qu'est ce que UML :

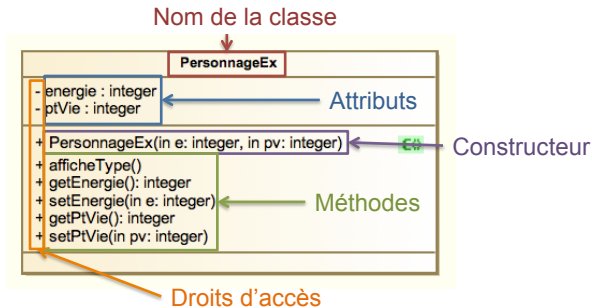
- Langage de modélisation graphique utilisé dans le développement logiciel et pour la conception orientée objet
- Cf. le cours de Génie Logiciel en option Informatique
- Propose 14 diagrammes qui permettent de modéliser le cycle de vie complet d'un projet logiciel

Pour le module POO :

Utilisation principalement d'un seul type de diagramme : le **diagramme de classe** ! Ces diagrammes permettent une représentation graphique des classes (attributs, méthodes, etc.) et des relations entre ces classes



## Classe : Notation UML



Convention *CamelCase* (en fait *lowerCamelCase*)

Le nom de la classe commence par une majuscule, le nom des méthodes et des attributs par une minuscule  $\Rightarrow$  utilisée en UML et en Java