Programmation Java
Introduction

Patrick Dezécache

ULCO - EILCO

November 9, 2015

Ressources

Contacts/Questions :

- Patrick Dezécache : patrick.dezecache@univ-littoral.fr

Bibliographie/webographie :

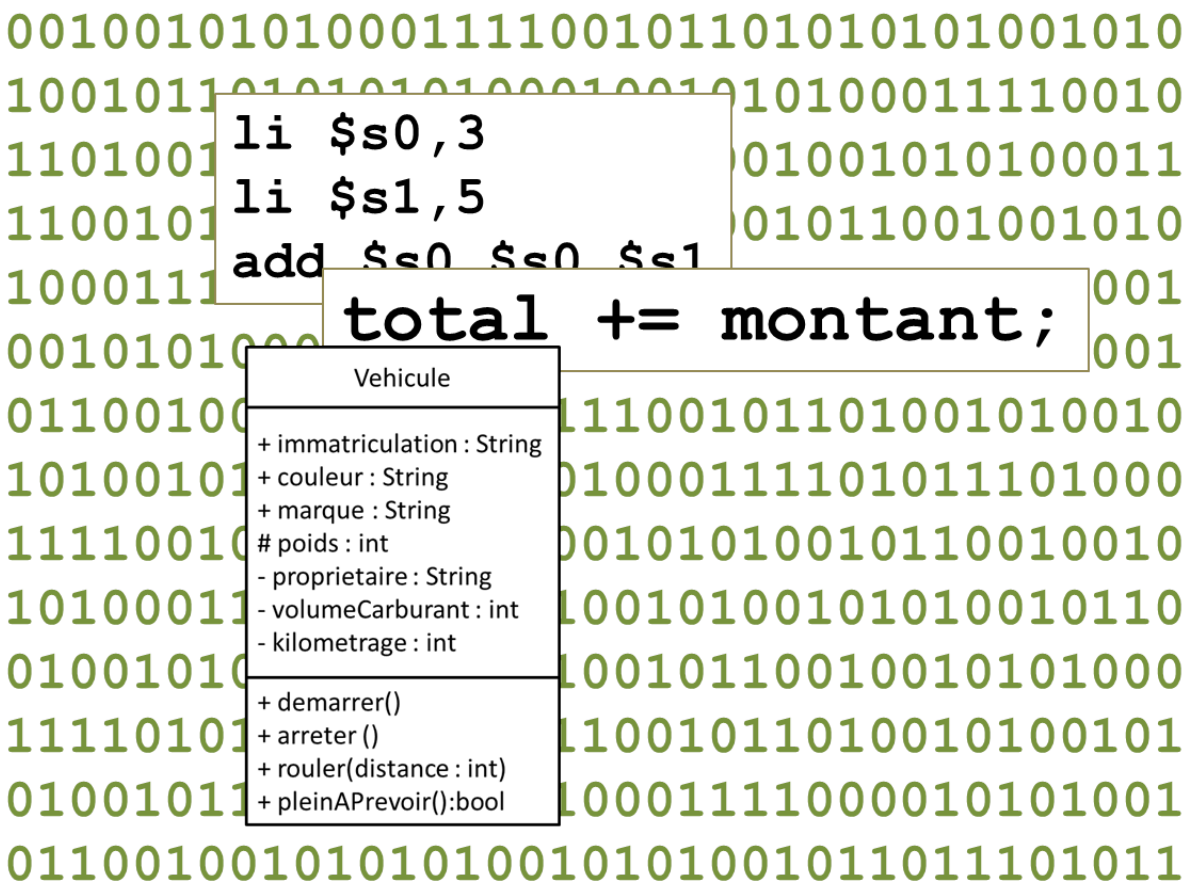
- Claude Delanny, Programmer en Java, Edition Eyrolle, 2004
- Kathy Sierra, Bert Bates, Head First Java, 2nd edition, O'Reilly Media, 2005 Head First Java, 2nd edition, e-book
- Bruce Eckel, Thinking in Java, téléchargeable sur le site http://www.bruceeckel.com ou http://greenteapress.com/thinkapjava
- The Java Language Specification (Java SE 8 Edition)
- API Java, Java Tutorial

Logiciels

- le JDK (*Java Development Kit*) : téléchargeable sur le site d'Oracle (http://www.oracle.com/technetwork/java/javase/downloads/index.htm
  - inclut le JRE (*Java Runtime Environment*), comportant la JVM (*Java Virtual Machine : chargeur et exécuteur de classes*) et l'API Java (*vaste collection de classes - $\sim$ aux bibliothèques C/C++, mais plus riche*)
  - le compilateur de classes
- un logiciel pour écrire le code source
  - un editeur de texte (Notepad++, Pspad, Blocnote)
  - un editeur éducatif pour Java (BlueJ, Jeliot)
  - un editeur spécialisé pour Java associé à UML(ArgoUML)
  - un EDI (*Environnement de Développement Intégré*): le plus complet, nécessite une durée non négligeable de prise en main, et intègre plusieurs outils : éditeur, compilateur, documentation (Eclipse, NetBeans)

Introduction

Introduction



Algorithmique

Algorithme

- ensemble d'actions, qui, appliquées à des données d'entrées, permettent d'obtenir un résultat
- précis, non ambigu, se termine en un temps déterminé
- objectif : faire exécuter par une machine programmable

Connaissance indispensable du mode de résolution :

```
lire le rayon,
calculer la circonférence (2 X PI X rayon)
ecrire la circonférence
```

Si le mode de résolution n'est pas acceptable, faire appel à une heuristique (*mode de résolution approché avec un temps d'exécution acceptable*)

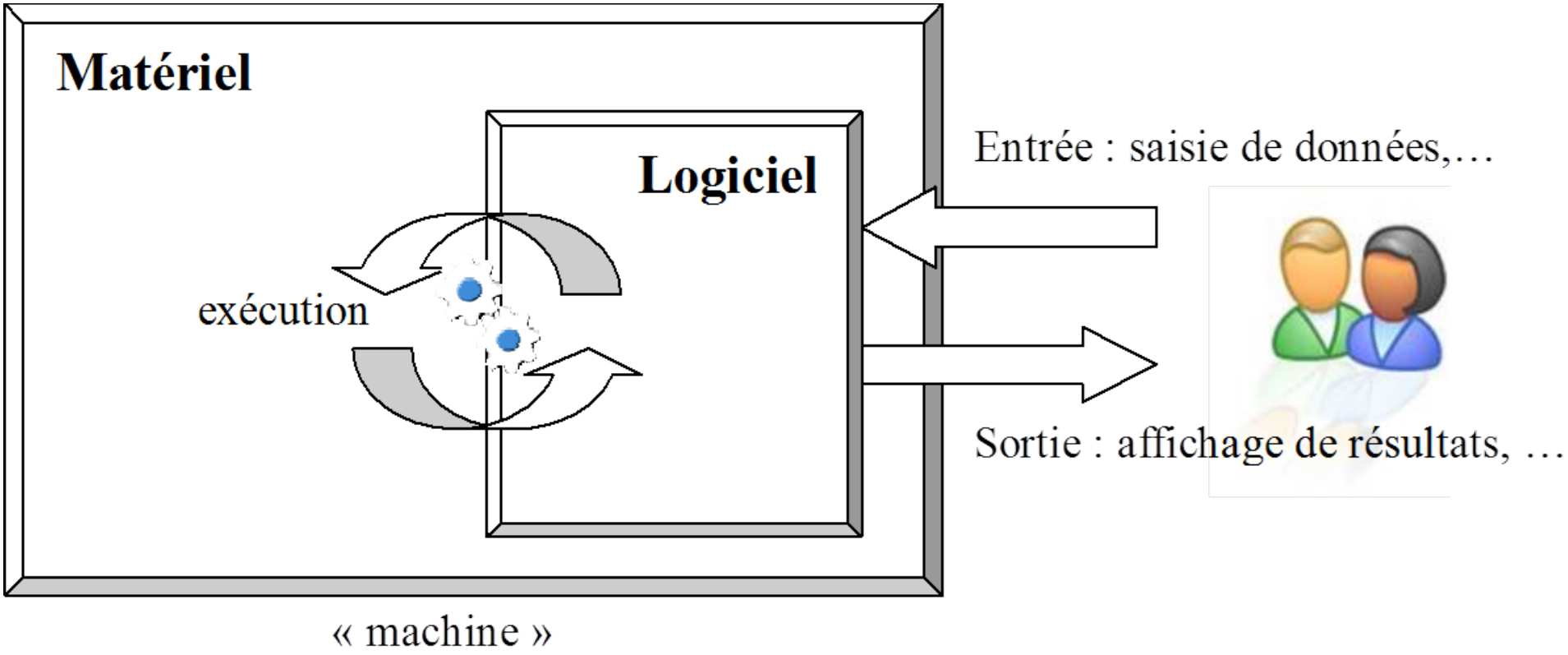
Algorithmique

Le pseudo-code, une des formes de l'expression de la résolution dans un langage (quasi-)naturel.

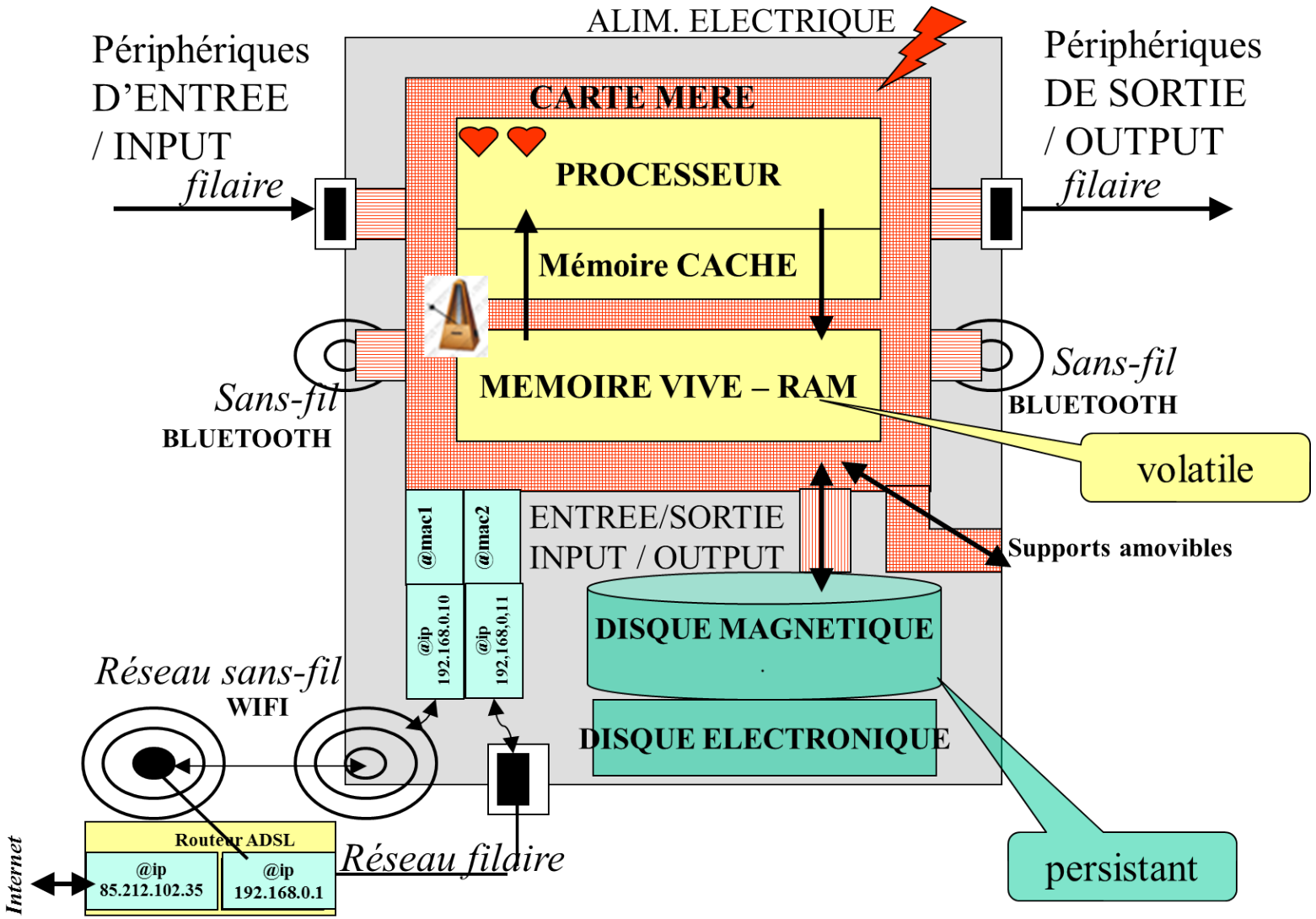
```
Algo calculerCirconférence
Données
    constante PI : reel <-- 3.1415927
    variables rayon : entier, circonférence : réel
Traitements
    lire rayon
    circonference <-- 2 *  PI * rayon
    ecrire circonference
```

D'autres formes d'expression graphiques.
Une fois l'algorithme validé, passage à la programmation afin d'automatiser l'exécution.

Architecture générale de l'ordinateur



Architecture générale de l'ordinateur



Unité centrale de traitement et mémoire vive

Au coeur de l'ordinateur, "mécanique" électronique :

- jeu d'instructions et registres du microprocesseur
- mémoire vive pour programmes et données en cours d'utilisation
- horloge de cadencement de l'exécution
- le programme, séquence d'instructions

Tout est stocké en binaire (langage machine) :

```
0010 0100 0001 0000 0000 0000 0000 0011
0010 0100 0001 0001 0000 0000 0000 0101
0000 0010 0001 0001 1000 0000 0010 0000
```

Pour "simplifier" l'écriture, l'hexadécimal :

```
0x24100003
0x24110005
0x02118020
```

Programmation en langage d'assemblage

Évolution :

- à chaque instruction, son code mnémonique
- à chaque registre, son code
- compilation : traduction d'un code source en code exécutable
- le code source est dépendant de l'architecture matérielle

Langage d'assemblage :

```
li $s0,3           #charger la valeur 3 dans reg. $s0
li $s1,5           #charger la valeur 5 dans reg. $s1
add $s0,$s0,$s1 #additionner $s0 et $s1, ranger dans $s0
```

cf. simulateur MARS

Langages de haut-niveau

Faire abstraction des contraintes matérielles

- s'approcher du langage naturel
- traduction du code source en langage d'assemblage, puis en code exécutable
- le code source devient "universel"

Exemple (Cobol74) :

```
COMPUTE total = total + montant
  ON SIZE ERROR MOVE 0 TO total.
```

Exemple (C/C++/Java) :

```
total += montant;
```

Langages de programmation

Multiplicité des langages de programmation

- types de problèmes à résoudre : gestion, calcul scientifique, web
- évolution "naturelle" (adaptation) : réponse à de nouvelles exigences
- paradigmes de programmation : différentes vues de l'esprit sur la manière de concevoir un programme
- modes d'exécution : interprété, compilé, semi-compilé

Paradigme procédural

Dans le paradigme procédural, structuré

- le logiciel est vu comme un ensemble de procédures
- la programmation impérative (succession d'ordres donnés à la machine)
- le programme est découpé en modules : procédures, fonctions
- des bibliothèques de fonctions réutilisables sont disponibles

Exemple, les fonctions de la bibliothèque math du langage C :

```
double sqrt (double x)
double pow (double x, double y)
```

Langages : Basic, Pascal, C, Cobol, Fortran, Basic, etc.

Autres paradigmes

Autres paradigmes

- objet : le logiciel vu comme une ensemble d'objets s'échangeant des messages (langages : Smalltalk, C++, Java)
- fonctionnel : le logiciel vu comme une fonction (langages : ML, Lisp, Scheme, Caml)
- logique : base de faits, règles, moteur d'inférence qui déduit de nouveaux faits pour répondre à des questions (langages : Prolog)
- déclaratif : le programme vu comme un ensemble de déclarations (langages : XSLT, SQL)

Paradigme objet

Apparu dans les années 60, il a pris son essor dans les années 90 et il est aujourd'hui le paradigme majeur du développement informatique des Systèmes d'Information

- le programme vu comme un ensemble d'objets
- un objet : entité du monde réel (matérielle ou immatérielle), il possède
  - une identité propre (qui le rend unique),
  - des *membres* :
    - *attributs* : états significatifs, ensemble d'attributs qui le décrivent à un certain moment de son existence
    - *méthodes* : comportements, ensemble d'opérations qu'il peut réaliser en réaction à un message (on parle d'invoquer/faire appel à une méthode)
- une classe : regroupe les caractéristiques communes à un ensemble d'objets

La classe représente un modèle, un patron permettant la création d'objets

Paradigme objet

Exemple de classes, description/état, comportement

- machine : numéro, état actuel (marche/arrêt), nombre d'unité produites, date de démarrage, date d'arrêt, démarrer, arrêter, produire une unité, calculer la durée de travail, indiquer son état
- bâtiment : adresse, état (ouvert/fermé), ouvrir, fermer
- client : numéro, nom, adresse, changer d'adresse
- transaction : numéro, date, compte à créditer, compte à débiter, montant, créer

Quels caractéristiques (description/état) et comportements pour :

- véhicule ?
- compte bancaire ?

Paradigme objets et notation UML

La notation UML (*Unified Modeling Language*), issue de l'OMG (*Object Management Group*)[1]

- un langage basé sur un metamodèle
- un ensemble de diagrammes qui permettent la représentation visuelle d'un système (*objet ayant une certaine autonomie*) selon différents angles
  - aspects statiques du système (description)
  - aspects dynamiques du système (interaction entre les objets)

La notion de classe décrit les objets d'un système :

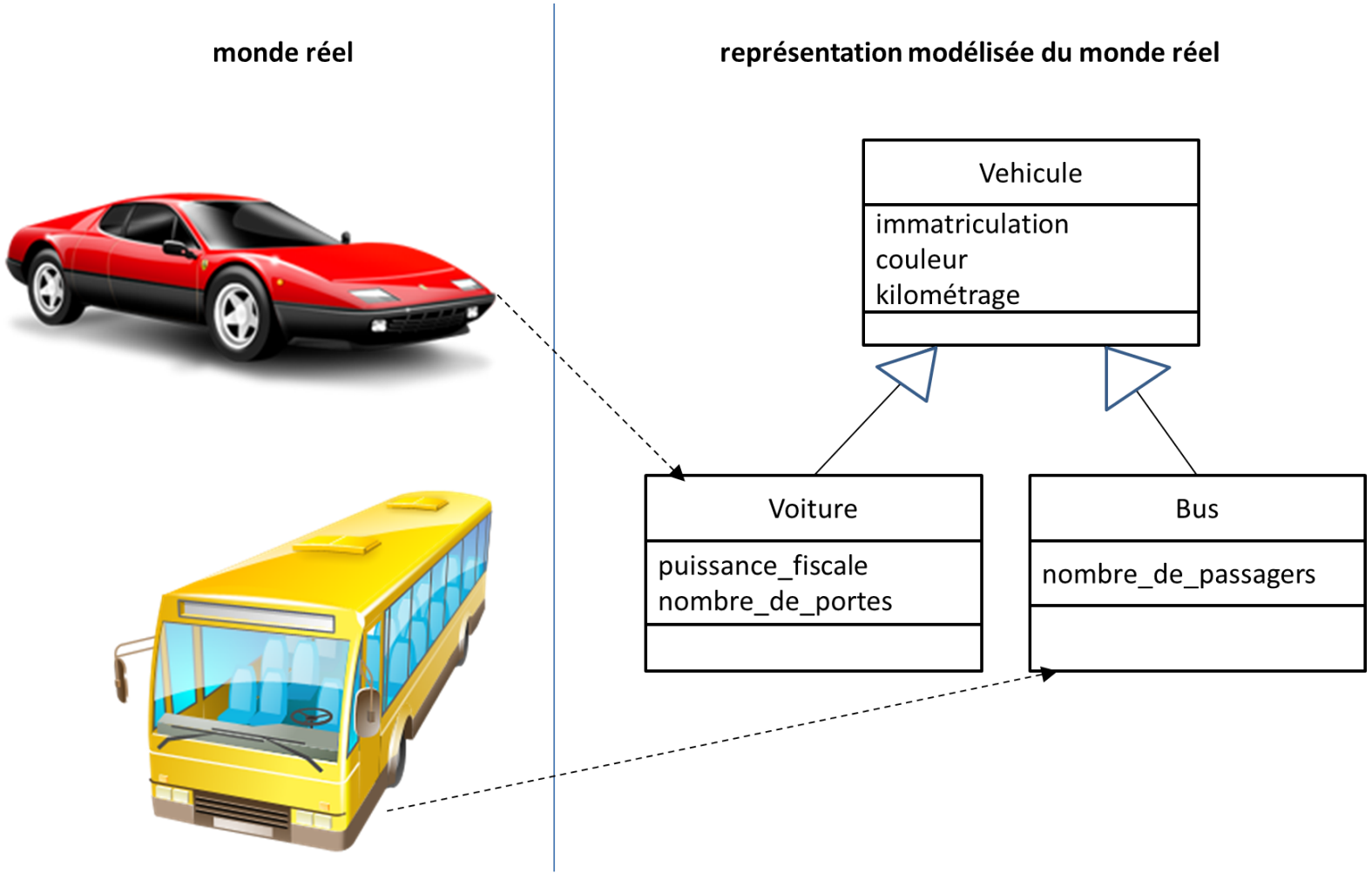
- classe, membres attributs et méthodes communs à une classe d'objets
- des relations entre classes
- objet, instance d'une classe, un exemplaire particulier

Le diagramme de classes représente l'aspect statique d'un système

[1] http://www.omg.org/, http://uml.free.fr/

UML modélisation du monde réel

L'objectif de ces diagrammes est la représentation du monde réel modélisé

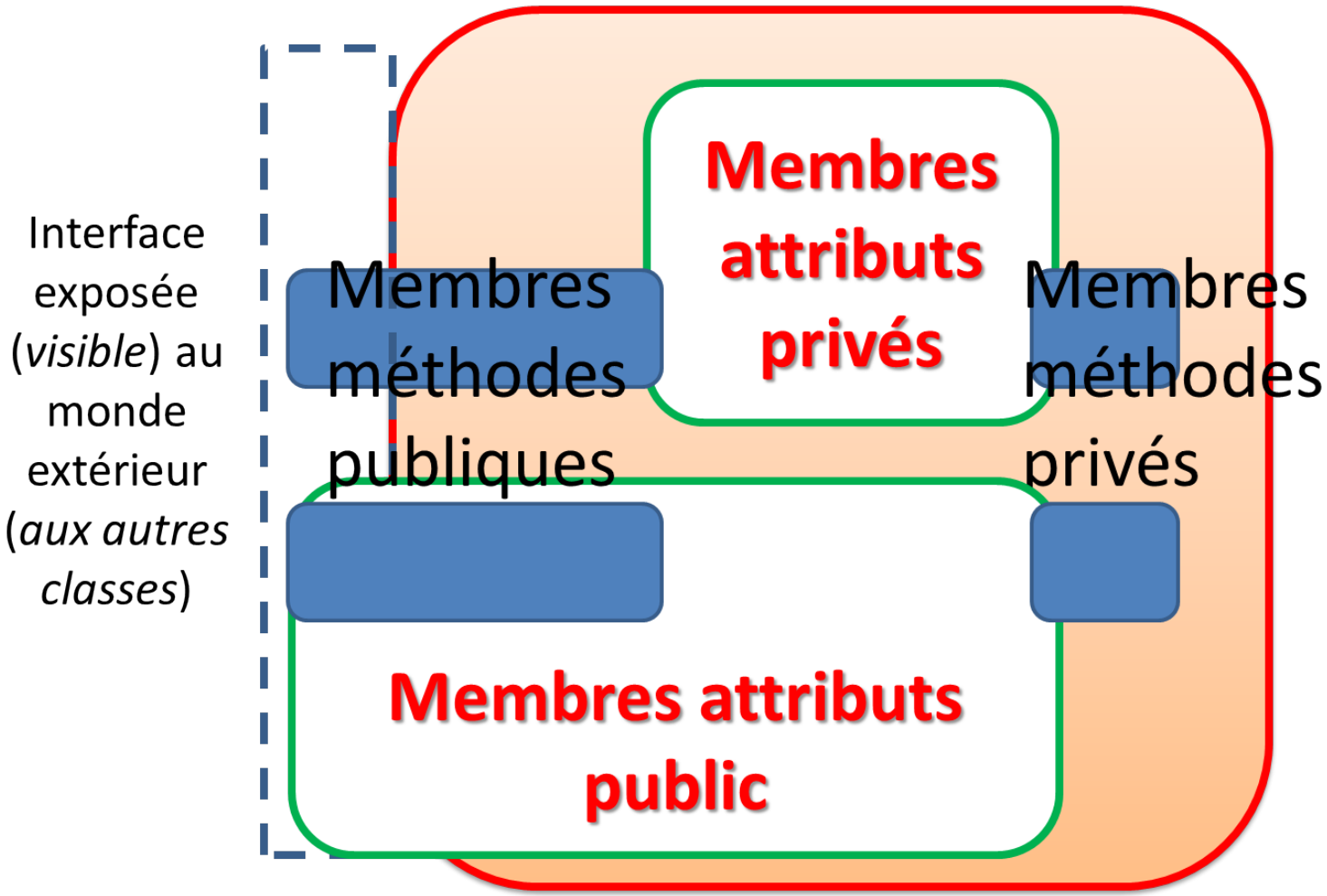


Encapsulation

Mécanisme de protection des membres d'un objet, en limitant leur visibilité/accessibilité afin de refuser et/ou contrôler toute opération d'accès ou de modification des membres

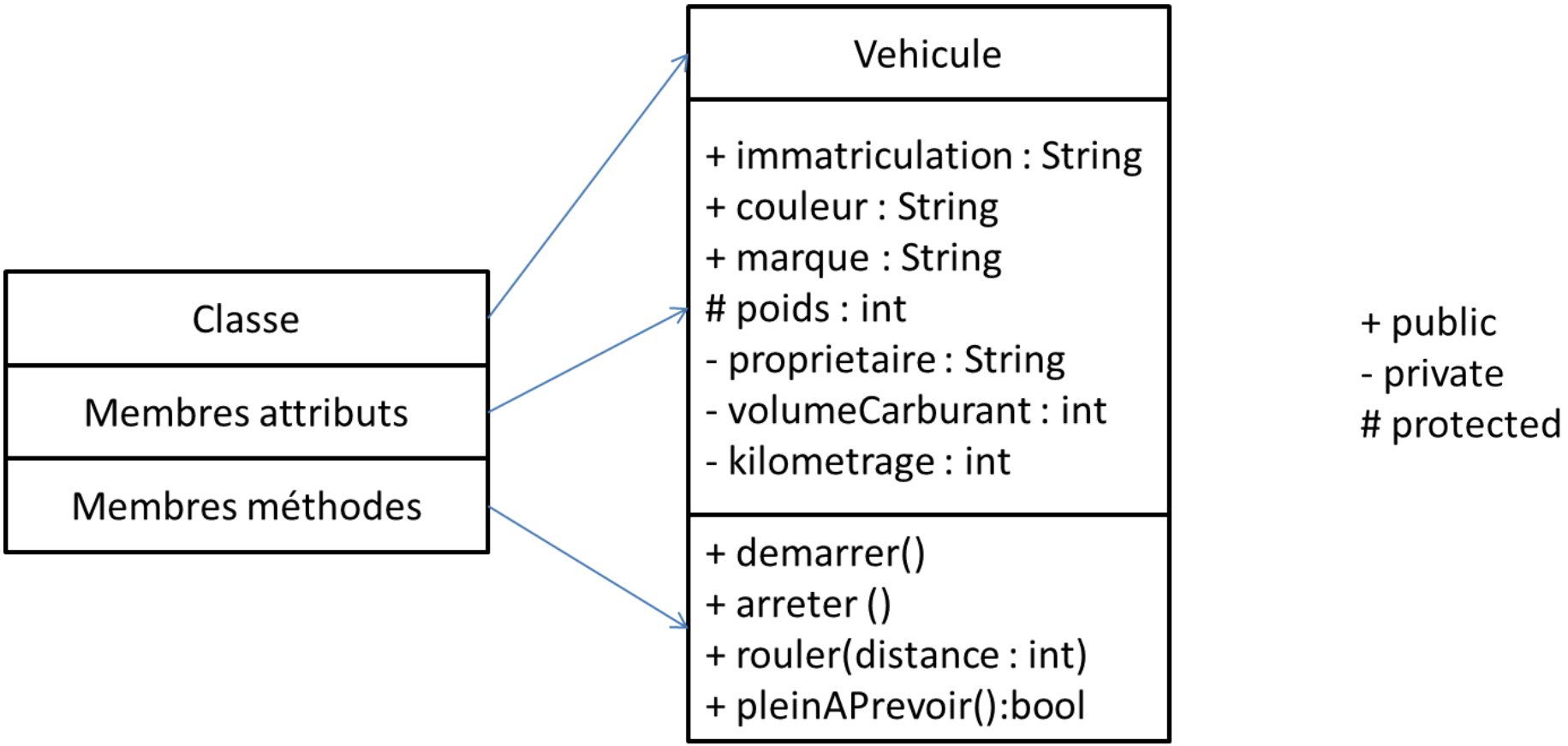
- privé : visible de la classe
- publique : visible/accessible de partout
- protégé : visible de la classe et des classes enfants
- package : visible des classes du même package

Encapsulation



Notation graphique UML

Diagramme de classe



(visibilité des attributs et des méthodes : notion d'encapsulation)

Héritage, polymorphisme

Héritage : capitalisation des membres communs dans une classe mère dont les classes filles héritent

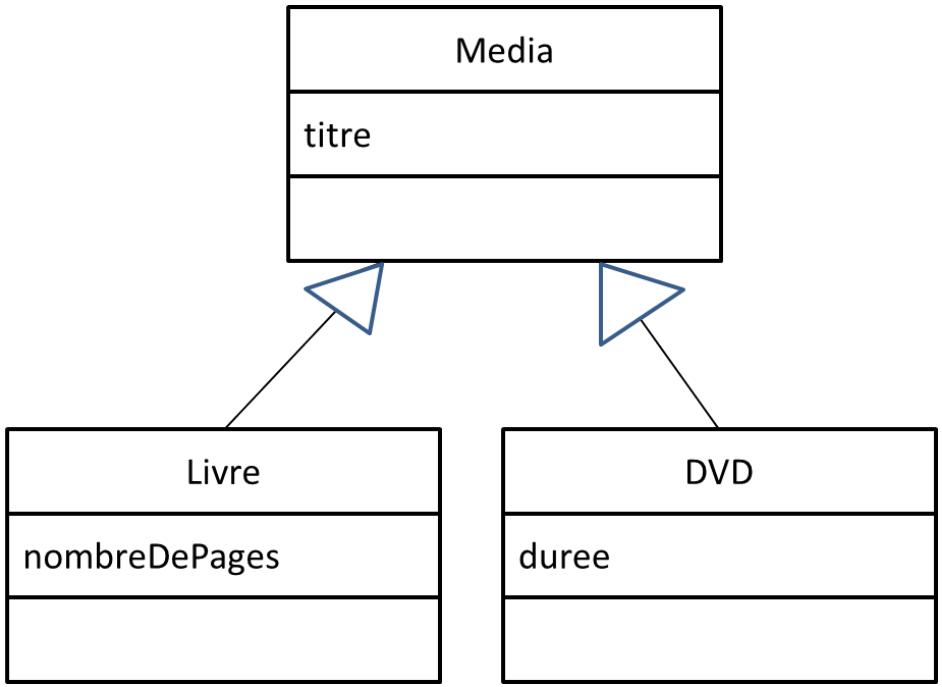
- classe générique (mère, super-classe) : propriétés et comportements communs
- classe spécifique (enfant, sous-classe) : propriétés et comportements spécifiques
- les classes 'enfant' héritent des membres communs visibles de la classe mère

Polymorphisme de redéfinition : possibilité de redéfinir des méthodes dans les classes 'enfant'

- les méthodes sont redéfinies dans les classes 'enfant' afin de décrire un comportement spécifique, différent du comportement de la classe mère
- les méthodes redéfinies ont la même signature (nom de méthode, nombre et types des paramètres)

Notation graphique UML

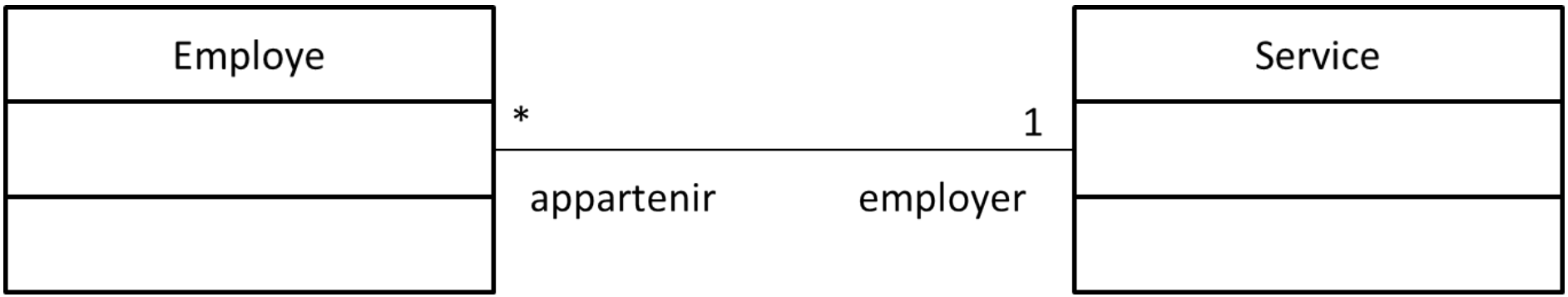
Diagramme de classe : héritage (spécialisation/généralisation)



- un livre est un média particulier : possède un titre comme tout média, mais sa spécificité est qu'il comporte un nombre de pages
- un DVD est un média particulier : possède un titre comme tout média, mais sa spécificité est qu'il comporte une durée

Notation graphique UML

Diagramme de classe : association entre 2 classes

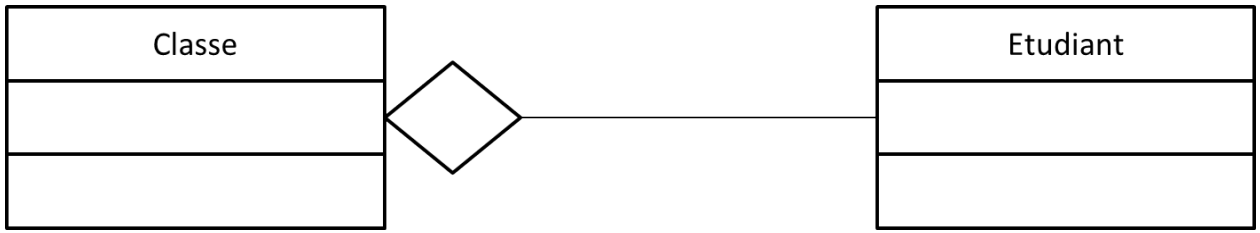


Les cardinalités quantifient les relations

- un service emploie des salariés (*)
- un employé appartient à un seul service (1)

Notation graphique UML

Diagramme de classe : agrégation

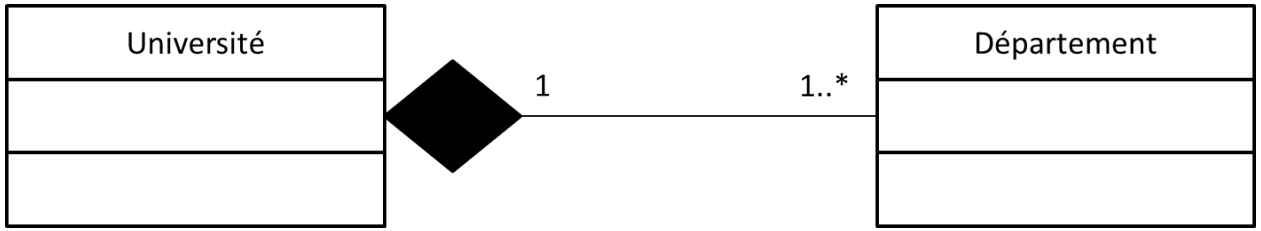


L'agrégat implique un lien faible entre 2 classes (lien ensemble/élément)

- une classe est un agrégat d'étudiants
- mais la vie d'un élève ne dépend pas de la classe (hors de la classe, l'étudiant a une vie !?), il peut exister sans être relié à une classe

Notation graphique UML

Diagramme de classe : composition



La composition implique un lien fort entre 2 classes (lien composé/composant)

- une université est composée de plusieurs départements (au moins 1)
- un département est relié à une université : si l'université disparait, le département disparait aussi

Notation graphique UML

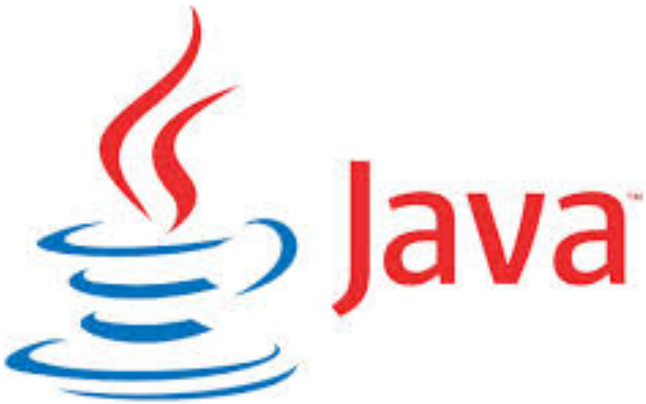
Exprimer la dynamique du système :

- Diagramme de collaboration : interactions entre instances
- Diagramme de séquence : exprime les interactions dans le temps

Des outils exploitent ces diagrammes pour générer le code informatique correspondant (en fonction du langage souhaité)

Java

Java



Java

- 1982, SUN Microsystem, serveurs et station de travail performants, OS Solaris
- 1995/1996, Technologie Java

- 1977, Oracle Corporation , SGBDR
- 2009, rachat de SUN par Oracle

Ressources Java sur le site d'Oracle

Java

- simple, OO, portable, distribué, multithread
- performant, robuste, sûr [2]

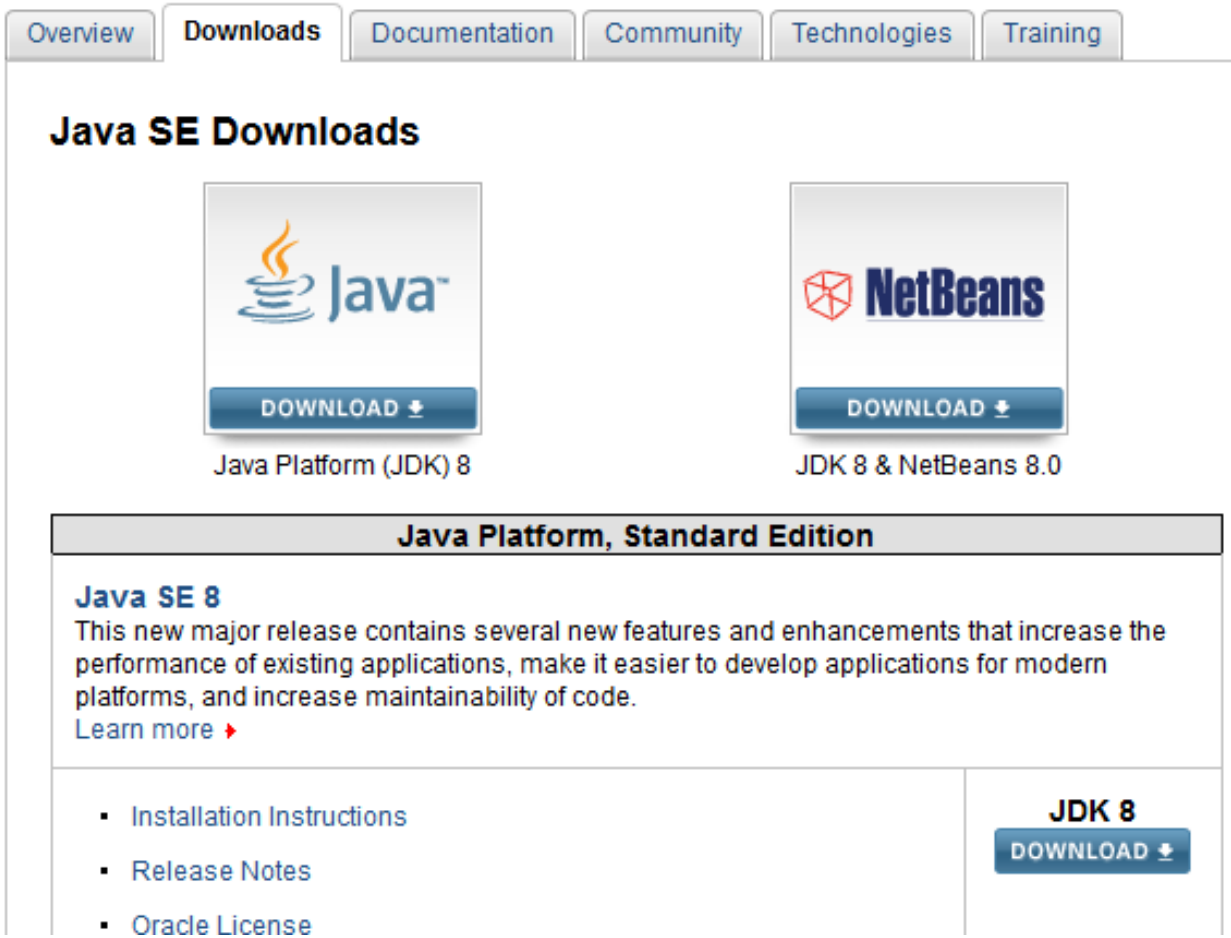
- apprentissage simple (basé sur C, mais sans pointeurs...)
- développement plus rapide et plus propre grâce à l'API Java (4 X moins de lignes de code qu'en C)
- indépendant de la plateforme matérielle (à condition d'avoir une JVM adaptée...)
- distribution du logiciel aisée (à condition d'avoir la JVM adaptée sur le site cible ...)

[2] des failles de sécurité sont régulièrement corrigées...

Java

où trouver Java ?

http://www.oracle.com/technetwork/java/javase/downloads/index.html



Java

Ancêtres ?

C (langage de base), C++ (objets), sans les pointeurs

Paradigme ?

orienté objet

Mode d'exécution ?

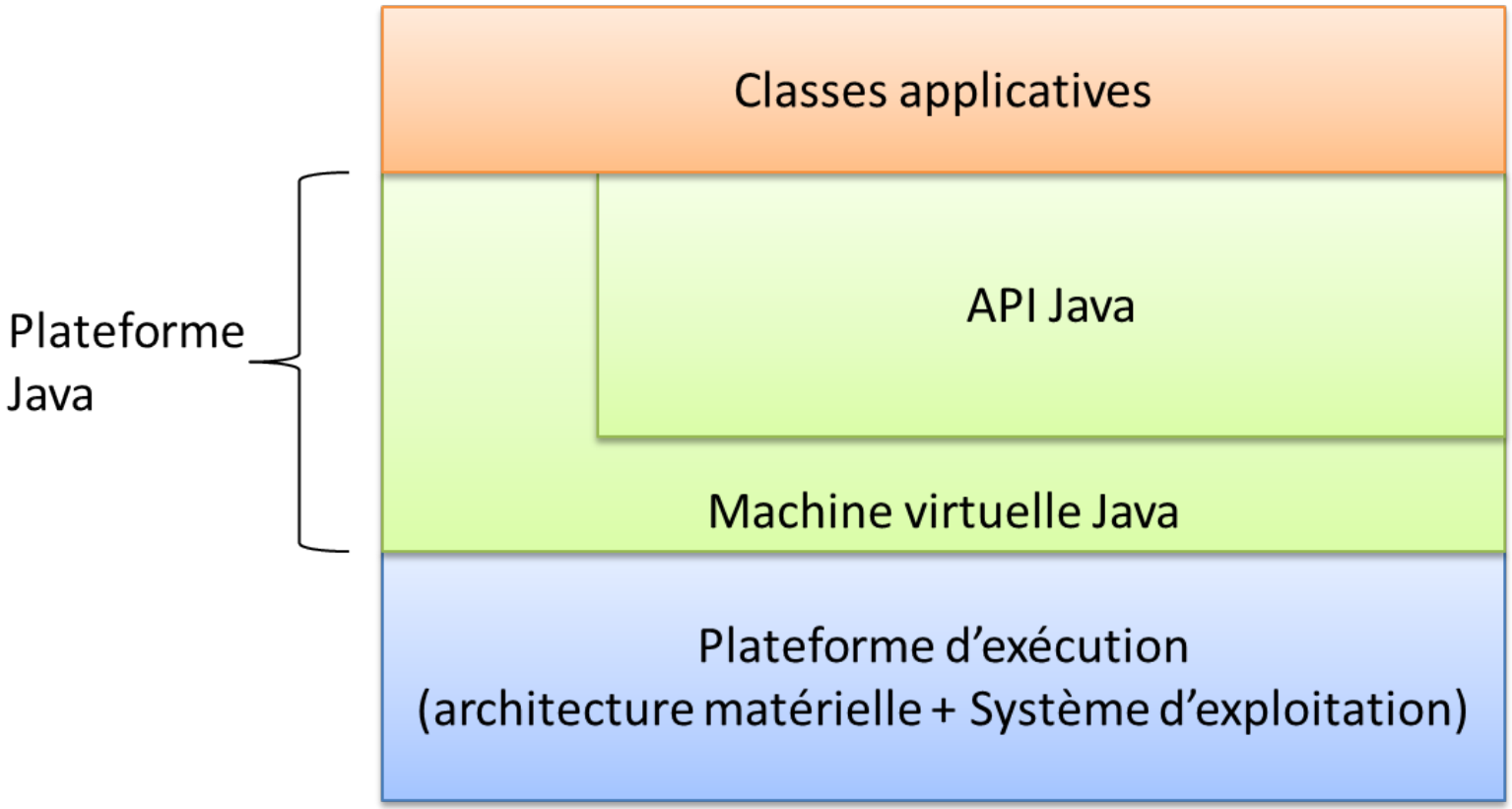
semi-compilé (byte code), exécuté par la JVM (portable), garbage-collector (ramasse-miettes)

Slogan

Write Once, Run Anywhere !

Plateforme Java

Plus qu'un langage, une plateforme



L'API Java : collection de classes et interfaces (*près de 4000*), organisées en packages (*environ 200*) et très bien documentée (*accès à la documentation Java indispensable pour programmer*)

Editions de la plateforme Java

Différentes éditions pour supporter différents types d'applications

- J2ME (micro) : applications embarquées sur assistants personnels, terminaux mobiles, portables
- **J2SE** (standard): ordinateurs de bureau
- J2EE (entreprise): serveurs d'entreprises

Chaque édition définit des API spécifiques.

Java

Des version des plateformes régulièrement éditées (*corrections, améliorations*) : la compatibilité ascendante est assurée (*un programme Java compilé avec une version antérieure peut être exécuté avec une version plus récente de la JVM*).

Historique des versions de Java :

version	date	nombre de classes de l'API
JDK1.0	mai 1995	env. 250
JDK1.1	février 1997	env. 500
J2SE 1.2	décembre 1998	
J2SE 1.3	mai 2000	env. 1840
J2SE 1.4	février 2002	env. 2500
J2SE 5.0	Sept.2004	3279 (générique, enum)
Java SE 6	Décembre 2006	3793
Java SE 7	Juillet 2011	3970, version licence GPL
Java SE 8	Mars 2014	4240

Chaque version complète l'API avec de nouvelles classes

Java

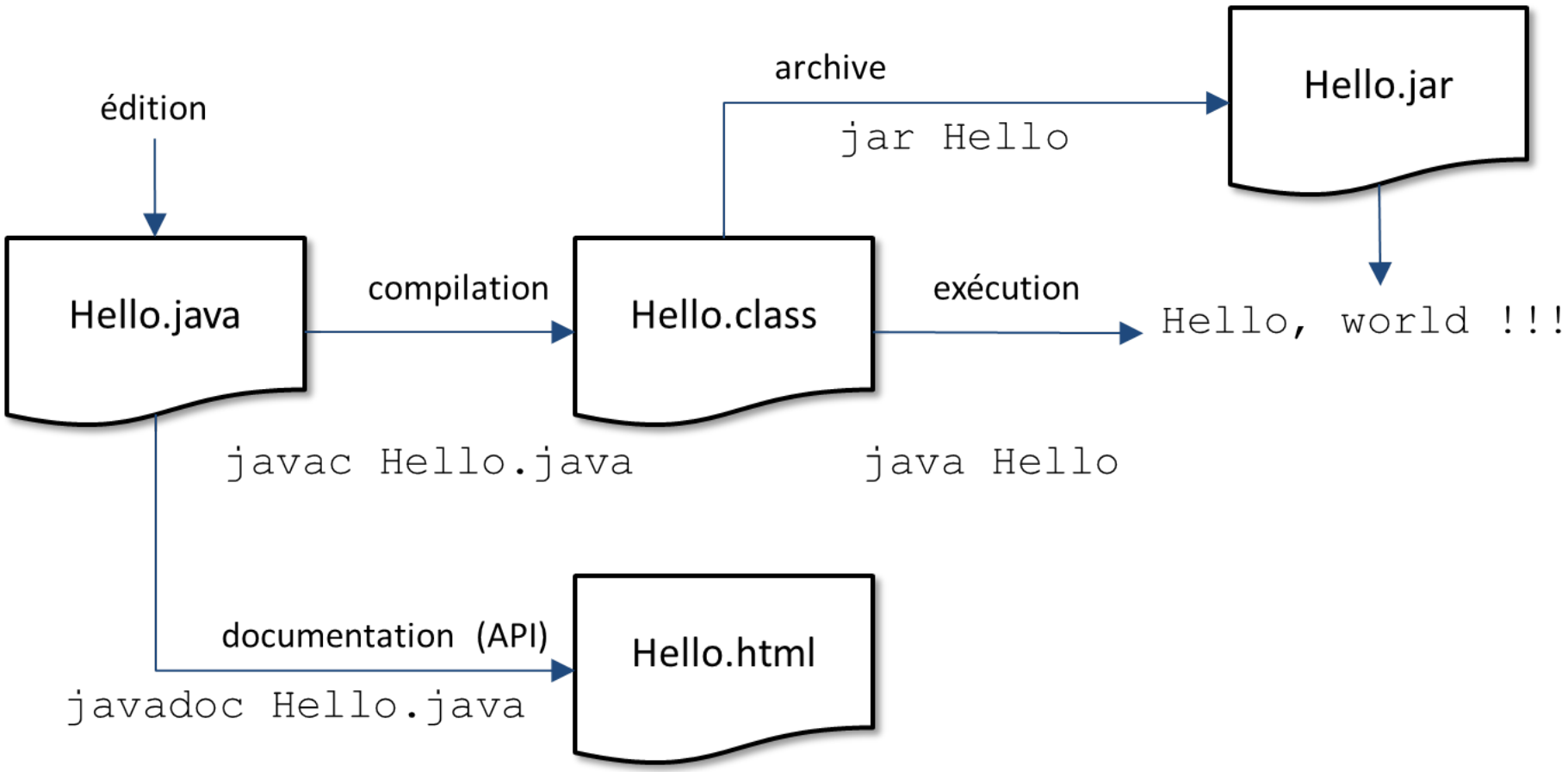
Etapes : du codage à l'exécution

1. Un code source Java (*correspond généralement à une classe Java*) est saisi dans un fichier texte brut : extension de fichier `.java`
2. le fichier est traduit dans une version semi-compilée (byte-code) : extension de fichier `.class`
3. la classe comportant une méthode `main` peut être chargée pour démarrer l'exécution

Une application comporte généralement plusieurs classes.
En complément :

1. la documentation de la classe peut-être générée sous forme d'une page web (extension de fichier `.html`)
2. les classes qui composent l'application peuvent être archivées sous forme d'une archive Java (Java ARchive) afin de simplifier la distribution du logiciel (extension de fichier `.jar`)

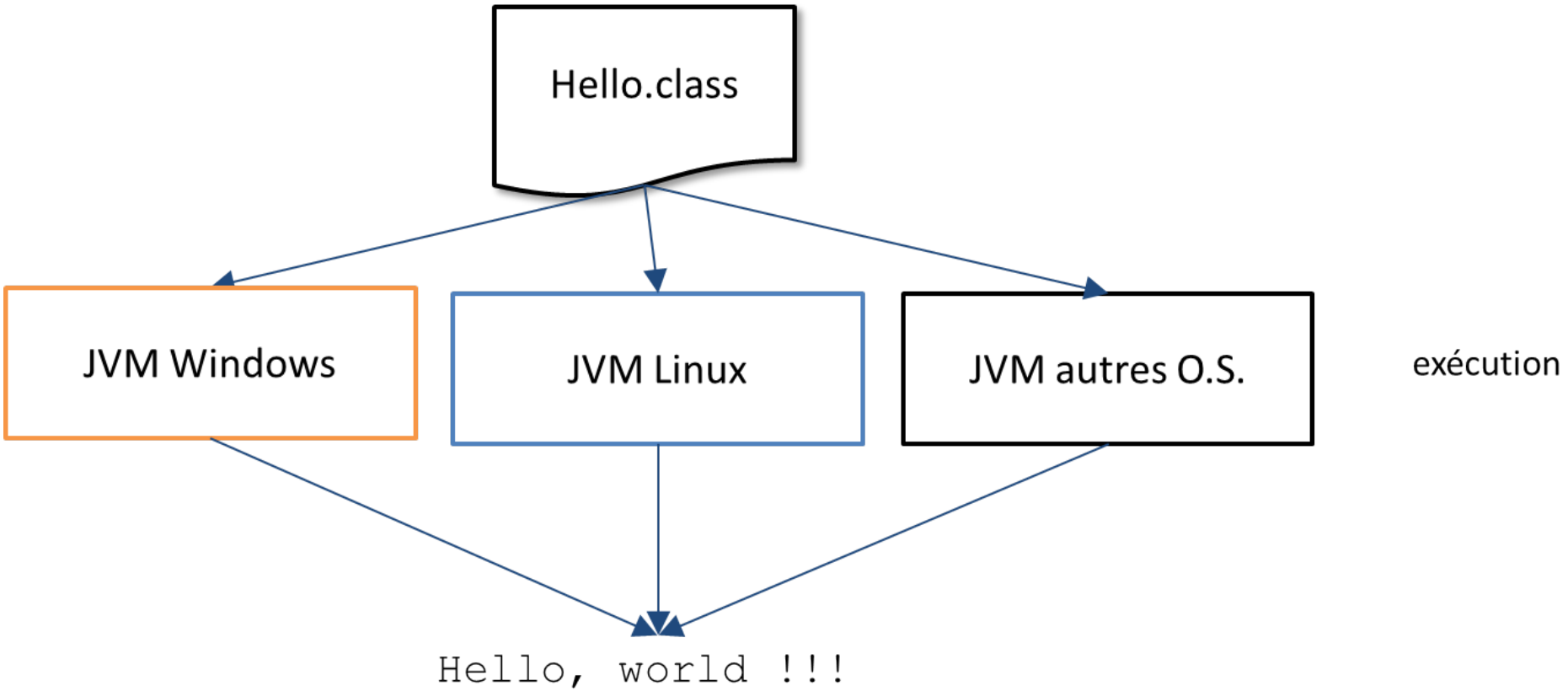
Java



La code source Java est transformé en bytecode Java, puis exécuté par la JVM

Java

Write Once (Compile Once), Run Anywhere



Java

Write Once (Compile Once), Run Anywhere

- Des machines virtuelles Java (*Java Virtual Machine*): Oracle/Sun(HotSpot), IBM, Microsoft (*répondent aux spécifications définies pour une JVM*)
- Des plateformes d'exécution : couple JVM et architecture logicielle d'exécution
- La machine virtuelle charge des classes (class loader), optimise les portions de codes (profiler), compile "à la volée" (*Just In Time compiler, traduction du bytecode en langage machine et exécution*)
- Elle se charge de la gestion des ressources mémoire grâce à son garbage-collector (ramasse-miettes)

Eléments du langage

- pas de structure d'écriture préétablie (*tout peut-être sur une ligne...lisibilité = indentation*)
- sensible à la casse
- identificateurs : lettres, chiffres, tiret bas (du 8)
- littéraux
- commentaires
- fin de "phrase" : ";"
- séparateurs
- mots clefs Java : types, expressions de calculs, instructions, structures de contrôle de l'exécution
- un style d'écriture Java

auxquels s'ajoute la collection de classes formée par l'API Java

Spécifications du langage (J2SE7)

Conventions de codage Java

Mots-clefs du langage Java

Ces mots-clefs ne peuvent pas être utilisés comme identificateur :

```
abstract  continue  for  new  switch
assert  default  goto  package  synchronized
boolean  do  if  private  this
break  double  implements  protected  throw
byte  else  import  public  throws
case  enum  instanceof  return  transient
catch  extends  int  short  try
char  final  interface  static  void
class  finally  long  strictfp  volatile
const  float  native  super  while
```

Types de données de base

Hérités du langage C, 8 types primitifs :

- 4 nombres entiers (signés): **byte**, **short**, **int**, **long**
- 2 nombres réels (valeur approchée) : **float**, **double**
- booléen : **boolean** (valeurs **true** ou **false**)
- caractère Unicode : **char**

Chaque variable doit être typée. Exemple :

```
char uneLettre='a';
byte unMois=5;
short uneAnnee=2012;
int unePopulation=65596325;
long populationMondiale=7058215211;
float posX=250.25f, posY=120f;
boolean unTest=true;
```

Depuis J2SE7, séparateurs possibles dans les nombres : `int unePopulation=65_596_325;`

Unicode

Types de données de base

Limite de capacité de stockage par type de donnée :

type	plage de valeurs	taille en octets
byte	-128 à +127	1
short	-32768 à +32767	2
int	-2 147 483 648 à +2 147 483 647	4
long	-2^{63} à $+2^{63}-1$	8
float	$-1.4 * 10^{-45}$ à $+3.4 * 10^{38}$	4
double	$4.9 * 10^{-324}$ à $+1.7 * 10^{308}$	8
char	65536 caractères	2
boolean	vrai ou faux	1

Types de données de base

"représentation des nombres réels approchée" ?
La conversion binaire des nombres réels est approchée à cause de la limite de capacité des nombres en mémoire.
Exemple :

```
double d1 = 0.01;
double d2 = 1.37;
double d3 = d1 + d2;
```

La valeur de d3 sera : 1.3800000000000001 ! et donc différente de 1.38 !
Cela implique que des calculs précis (montants, par exemple) utiliseront un type entier plutôt qu'un nombre réel !

Structures de données : tableaux

Hérités du langage C, stocke un ensemble de données de même type

- le nombre d'éléments est déterminé
- accès par indice, de 0 et nombre d'élèments - 1
- opérateur **new** : création effective du tableau

Exemple :

```
int[] tableauEntier = {0,1,2,3,4,5,6,7,8,9};
char[] tableauCaractere = {'a','b','c','d','e','f','g'};
String[] tableauChaine = {"paris", "lille", "nice"};
int [] tableauEntier;
tableauEntier = new int[6];
int [] premierTableau, deuxiemeTableau;
int[][] matrice=new int[5][6];
```

Le tableau est un objet : propriété `length`, cf. API java.util.Arrays
Sa taille peut être déterminée à l'exécution à partir d'une variable saisie, par exemple.
Java offre des classes plus puissantes que les tableaux de base.

Opérateurs et expressions

Hérités du langage C, notion de priorité des opérateurs (*parenthèses pour contrarier une priorité d'exécution*)

- arithmétiques : résultat numérique

  `+,-,/,*,%`

- relationnels : résultat booléen

  `==,!=,>,>=, <, <=`

- logiques : résultat booléen

  `||,&&,!`

- manipulation de bits :

  `&(et),|(ou),^ (oux), ~ (non) , <<, >>, >>>`

Exemple d'expressions :

```
(a * x * x + b * x + c )
((nombre * 2)==(nombre + 2))
```

Opérateurs binaires

Exemples :

expression	binaire	
a	1100	soit 12 en décimal
b	1010	soit 10 en décimal
a & b	1000	a et b, pour dé-positionner et tester
a |b	1110	a ou b, positionner
a ^b	0110	a oux b
~a	0011	non a
a << 1	1000	multiplier par 2
a >> 2	0011	diviser par 2*2

Par exemple : 4 conditions peuvent être codées soit en utilisant 4 booléens, ou 4 bits d'un octet.

Affectations et affectations composées

Hérités du langage C

- affectation :

  =

- (pre/post) incrémentation/décrémentation :

  ++,--

- affectation composée :

  +=,-=,*=,/=,%=, &=, ≙, |=,

Exemple :

```
unTest = ( a > b ); // unTest de type boolean
System.out.println(a++ +" et "+ ++a); // affiche '5 et 7'
    (a = 5 au depart)
nb*=2; // nb est un des types numeriques
```

Structures de contrôle - exécution conditionnelle

ou test, hérité du langage C

- **if () {;} else {;}** : test avec/ou sans alternative:

```
if (condition à tester) {bloc si vrai;}
else {bloc sinon;}
```

- **if () {;} else if () {;} else {;}** : tests multiples :

```
if (condition à tester) {bloc si vrai;}
else if (condition à tester) {bloc si vrai;}
else {bloc sinon;}
```

- **() ? : ;** : expression conditionnelle (opérateur ternaire):

```
(condition à tester) ? {valeur si vrai;} : {valeur sinon;}
```

```
if (nombre<5) {nombre=0;} else {nombre*=2;}
nombre = (nombre<5) ? 0 : nombre*2;
```

Structures de contrôle - exécution conditionnelle

simplifier un test de plusieurs valeurs d'une même variable entière

- **switch()** {**case : break;** } : test de cas :

```
switch (valeur à tester)
    { case litéral1: bloc pour ce cas...; break;
      default: ...; break;
    }
```

```
switch (jourSem) {
    case 6:
    case 7:
      week_end = true; break;
    default:
      week_end = false;
}
```

J2SE7 autorise le test de variables de classe String

Structures de contrôle - exécution répétée

ou boucle, ou itération, hérité du langage C

- **for** : nombre de répétitions déterminée :

  `for(decl/init;condition de poursuite;incrémentation) {bloc`

- **while** :nombre de répétitions indéterminée 0 à n fois:

  `while(condition de poursuite) {bloc à répéter}`

- **do while** :nombre de répétitions indéterminée 1 à n fois :

  `do {bloc à répéter} while(condition de poursuite);`

- (nouveau) répétition pour chaque valeur d'une collection d'un certain type de donnée (*tableau, liste*)

  `for (type variable : collection de type) {bloc à répéter}`

Maitriser la condition de poursuite de boucle

```
int compteur = 1;
while (compteur<=10) {compteur++;}
for (int compteur = 1;compteur<=10;compteur++) {;}
```

Structures de contrôle - exécution répétée - équivalences

- **for (;;) {;}**

```
for (i = 1; i <= NB; i++) {
   j++; // traitement
}
```

- **while () {;}**

```
i=1;
while (i <= NB) {
   j++; // traitement
   i++; // incrementation
}
```

- **do {;} while ();**

```
i=1;
if (i <= NB) {
   do {
      j++; // traitement
      i++; // incrementation
   } while (i <= NB);
}
```

Structures de contrôle

Instructions permettant de quitter une structure de contrôle

- **break** : quitte définitivement la structure de contrôle et reprend l'exécution à l'instruction qui suit

```
for (i = 0; i < 12; i++) {
    ...
    if (i==7) break;
    ...
}
```

- **continue** : continue à l'itération suivante

```
for (i = 0; i < 12; i++) {
    ...
    if (i==7 || i==8) continue;
    ... pas de traitement pour 7 et 8
}
```

- **return** : quitte la procédure en cours

Un premier programme

Toute application Java est composée d'un certain nombre de classes.

Une application nécessite un point d'entrée (la ligne où doit commencer l'exécution) : une des classes qui la composent devra comporter la méthode 'main', définie comme 'static' (*elle peut être appelée directement sans instance de classe*); c'est cette méthode qui sert de point d'entrée. La méthode 'main' est invoquée après que la classe ait été chargée en mémoire.

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

♢ cf. classe HelloWorld version 1

Méthodes

Procédures et fonctions qui implémentent les traitements associés à des objets (*en d'autres termes : moyens de solliciter un objet*). L'appel d'une méthode d'un objet est vu comme l'émission d'un message vers un objet.

Définition d'une méthode :

- sa signature
  - modificateur(s) d'accès : public, private, protected
  - autres modificateur : static, etc.
  - type de valeur retournée ou `void`
  - son nom
  - entre parenthèses, liste des paramètres attendus
- entre accolades, définition du corps de la méthode
  - déclarations locales
  - instructions

Méthodes - modificateur static

Des modificateurs permettent d'apporter des précisions sur les propriétés de la méthode (ou un autre membre) :

- Le modificateur 'static' définit un membre comme pré-existant pour une classe donnée : c'est un membre de classe (attribut ou méthode).
  - Ainsi la méthode 'main' d'une classe est définie comme 'static'
- Sans modificateur 'static', un membre est dit "d'instance" : son existence sera conditionnée à celle d'une instance de classe

♢ cf. classe HelloWorld version 2
♢ cf. classe MonProgramme

Méthodes

Passage des arguments : toujours par valeur

- par valeur pour les types de base
- par valeur de la référence pour les objets

Ces valeurs sont copiées dans les paramètres de la méthode appelée.

Deux formes de méthodes

- **void** : sans valeur de retour, forme d'instruction complexe (procédures)
- avec un type de valeur de retour, type de base ou classe (*peuvent participer à des expressions*) (fonctions)

Java accepte les appels récursifs.

Méthodes et polymorphisme

Il peut exister plusieurs méthodes visibles et portant le même nom, mais elles auront des formes différentes (polymorphisme)

- Polymorphisme de surcharge (anglais overload) : la signature des méthodes est différente

```
class Calculatrice {
    ...
    public integer addition(int a, int b) { return a+b
        ;}
    public double addition(double a, double b) { return
        a+b ;}
    ...
}
```

- Le polymorphisme de redéfinition (anglais override) : la signature est identique, dans le cas de l'héritage de classes

Traitement des exceptions - structures de contrôle

Structure de contrôle de l'exécution qui permet de capturer les évènements exceptionnels (exceptions) : c'est le mécanisme d'interception des erreurs d'exécution en Java : **try** {;} **catch()** {;} **finally** {;}

Sans l'utilisation de ce mécanisme, en cas d'erreur, arrêt brutal

```
try {bloc à exécuter}
catch  (type variable) {bloc de traitement de l'exception}
finally {bloc à exécuter toujours}
```

Exemple (interception d'une division par 0):

```
int a = 4, b = 0;
try { a = a / b; }
catch (Exception e) {
    System.out.println("Division par zero !");
    e.printStackTrace();//sortie de la pile d'appels
}
finally { a = 0; }
```

Exceptions

L'exception est un évènement exceptionnel (*généralement signe d'une anomalie de fonctionnement*) déclenché lors de l'exécution d'un programme

- l'exception est un objet instancié lors du déclenchement
- l'instruction **throw** déclenche une exception
- une classe dans laquelle des exceptions sont déclenchées doit être déclarée comme **throws** suivi du type d'exception
- elle peut être utilisée pour vérifier les pré- et post- conditions des méthodes
- elle peut être utilisée pour tester les invariants de classe

```
void controler(int unNombre) {
  if (unNombre < 1 || unNombre > 9)
    throw new IllegalArgumentException("incorrect :"+
        unNombre );
}
```

Assertions

une assertion est une instruction permettant de tester/valider des hypothèses

- points de vérification dans un programme
- expression booléenne qui doit être vraie
- peuvent être activées ou désactivées, donc :
  - à éviter dans le test des paramètres
  - à séparer du code actif d'une méthode

Dans l'exemple ci-dessous, le passage par l'assertion est, à priori, impossible : **assert** évalue l'expression logique qui suit et lève une exception dans le cas où elle est fausse.

```
void uneMethode() {
  for(i=0;i<12;i++) {
     if (i>10) return ;
     }
     assert false; // leve une exception si on passe ici !
}
```

Commentaires de documentation

Les formes de commentaires ligne et bloc sont utilisable en Java.
Certains commentaires particuliers peuvent alimenter automatiquement la construction de la documentation d'une classe sous forme de pages Web, de la même forme que la document de l'API Java.
Une convention d'écriture des paramètres :

```
/**
  * un commentaire...
  */
```

Des "tags" identifient des éléments précis :

- @params : pour décrire les paramètres attendus d'une méthode
- @return : pour décrire la valeur retournée par une fonction

Commentaires de documentation

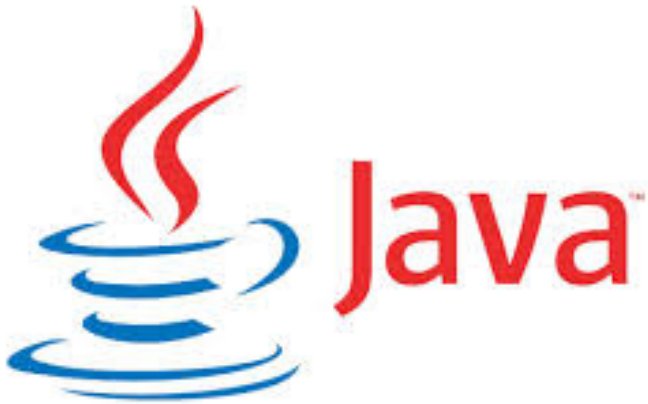
Exemple du fichier Etudiant.java :

```
/**
  * Etudiant : décrit un étudiant...
  */
class Etudiant {
   private String nom;
 /**
   * getNom : retourne le nom de l'étudiant
   * @return nom de l'étudiant
   */
   public String getNom() {return nom;}
}
```

L'outil **javadoc** génère la documentation de l'API des classes.

POO en Java

POO en Java



Classes et objets

abstraction des objets du monde réel, propriétés et comportements

- une classe : des propriétés et des méthodes spécifiques à une classe d'objets
- un objet, une instance d'une classe : opérateur **new** pour créer une instance de classe
- méthode(s) constructeur(s) des instances
- par défaut, une classe hérite de la classe **Object**
- un programme : ensemble de classes

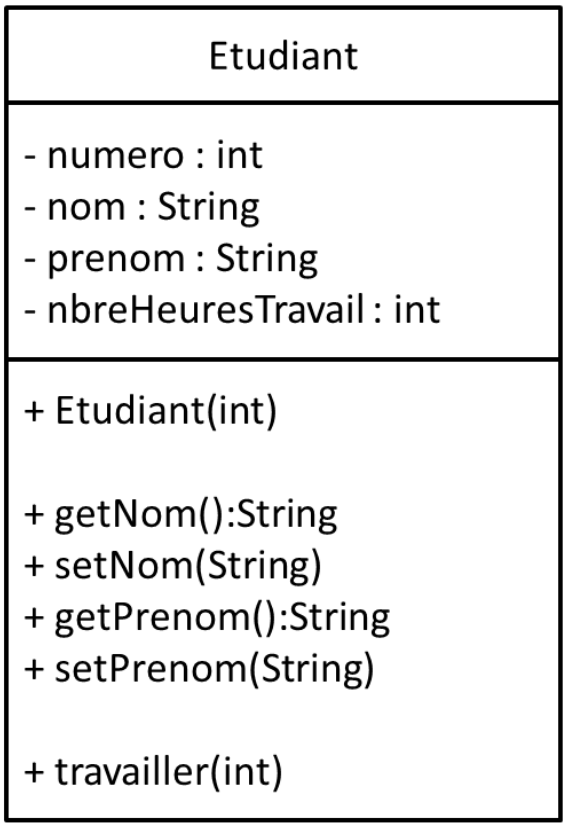
Ainsi, une classe "Etudiant.java" sera décrite ainsi :

```
class Etudiant {
    membres attributs : les propriétés décrivant un etudiant
    membres méthodes constructeur : méthodes d'initialisation
         de l'état initial d'un objet de type etudiant
    autres méthodes : les comportements accessibles à un etudiant
}
```

Classes et objets

classe Etudiant : une classe = un fichier, nom de la classe = nom du fichier[3]

Classe Etudiant ⟷ Fichier « Etudiant.java »




```
class Etudiant {
  // membres attributs
  private int numero;
  private String nom;
  private String prenom;
  private int nbreHeureTravail;

// méthode constructeur
  Etudiant(int n) {  }
  // autres méthodes
  public String getNom() {  }
  public void setNom(String n) {  }
  public String getPrenom() {  }
  public void setPrenom(String n) {  }

  public void travailler(int n) {  }
}
```

Classes et objets

Membres de classe

- le modificateur **static** définit un membre de classe
- il ne sera pas instancié
- on pourra invoquer ce membre (attribut ou méthode) en précisant le nom de la classe
- les méthodes statiques ne peuvent accéder aux membres d'instances

Membres d'instances

- chaque objet instancié contient une copie des membres d'une classe (sauf **static**), avec ses propres valeurs pour les membres attributs
- l'objet possède un identificateur unique
- la variable de déclaration fait référence à cet objet unique
- **this** fait référence à l'objet à lui même

Invocation d'une méthode d'instance :
$< nomd'objet >.< nomdemethode >$(arguments)

Classes et objets - méthode main

méthode publique (*accessible de tous*), et statique (*méthode de classe, invocable sans instance de classe*)

```
class MonProgramme {
    public static void main(String [] args) {
        //... faire quelque chose
    }
}
```

ou bien, pour permettre la définition de variables d'instances au lancement

```
class MonProgramme {
    //...variables d'instances ...
    private void executer() {
      //... instancier/initialiser les variables d'instance
      //... faire quelque chose
    };
    public static void main(String [] args) {
        MonProgramme p = new MonProgramme();
        p.executer();
    }
```

Classes et objets - constructeur

Méthode spécifique, le constructeur :

- porte le même nom que la classe
- est la méthode appelée lors de l'instanciation d'un objet
- permet d'initialiser les variables d'instance (=l'état de l'objet à sa naissance)
- permet de faire appel au constructeur des classes mères (méthode **super()**)
- si plusieurs constructeurs sont définis, recherche automatique du constructeur dont la signature correspond aux arguments d'appel

♢ cf. Chatterbox avec constructeurs

Classes et objets - accesseurs

Méthodes spécifiques, les accesseurs (getter/accesseurs et setter/mutateurs) permettent d'ouvrir un accès sélectif à des membres privés afin de

- contrôler l'accès aux attributs et aux méthodes sensibles
- contrôler la modification du contenu des membres attributs

Leurs signatures sont les suivantes :

- accesseur : **public** < *typedumembre* > get< *nomdumembre* >()
- mutateur : **public void** set< *nomdumembre* >(< *typedumembre* > p)

♢ cf. Chatterbox avec accesseur et modifieur (mutateur)

Cycle de vie des objets

Cycle de vie des objets

- l'objet apparait dès qu'il est instancié
- il vit tant qu'une référence existe vers lui
- à partir du moment où plus aucune référence y est faite, l'objet peut être détruit par le mécanisme de ramasse-miette (garbage collector)
- les ressources qui étaient référencées par l'objet sont libérées

Cycle de vie des objets

Exemple de classe Etudiant

- membres attributs d'instance
- membre constructeur

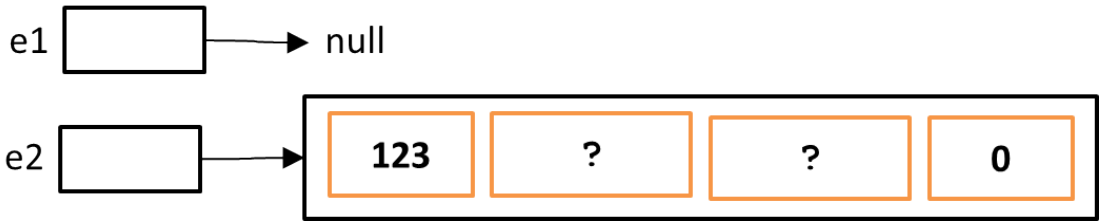
```
public class Etudiant {
    int numero;
    String nom;
    String prenom;
    int nbreHeureTravail;
    public Etudiant(int n) {
        numero = n;
        nbreHeureTravail = 0;
    }
    // ...
}
```

Cycle de vie des objets

Déclaration, instanciation, appel du constructeur

- déclaration d'une variable e1 de classe Etudiant : e1 ne référence aucun objet
- déclaration d'une variable e2 de classe Etudiant et instanciation d'un objet Etudiant de numéro 123 : e2 reçoit une référence vers cette instance

```
Etudiant e1;
Etudiant e2 = new Etudiant(123);
```

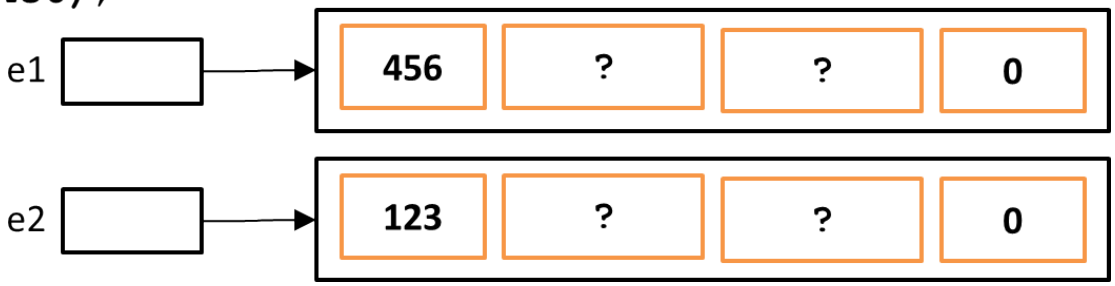


Cycle de vie des objets

instanciations, affectations

- Instanciation d'un objet Etudiant de numéro 456 : e1 reçoit maintenant une référence vers cette instance

```
e1 = new Etudiant(456);
```

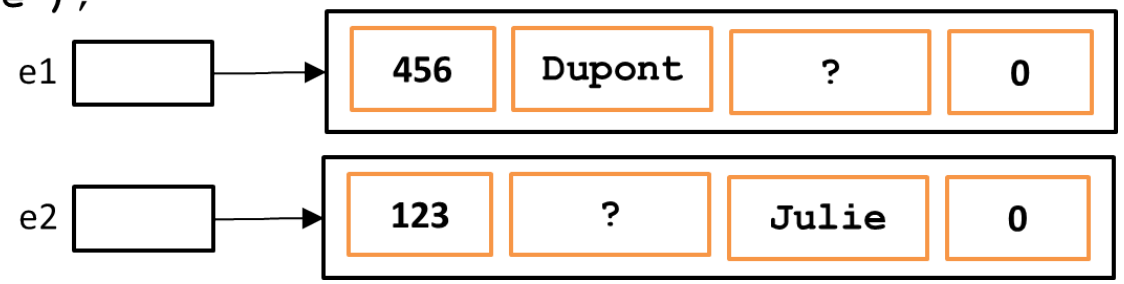


Cycle de vie des objets

Invocation des méthodes d'instances

- invocation de la méthode `setNom` de l'objet référencé par e1
- invocation de la méthode `setPrenom` de l'objet référencé par e2

```
e1.setNom("Dupont");
e2.setPrenom("Julie");
```



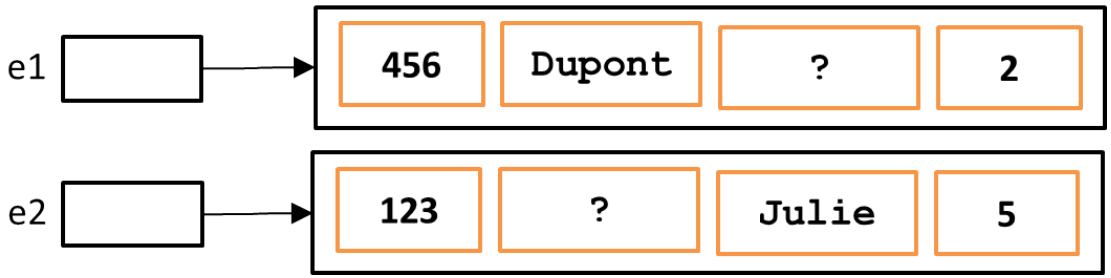

```
class Etudiant {
    // ...
    public void setNom(String unNom) {
        nom = unNom;
    }
    public void setPrenom(String prenom) {
        this.prenom = prenom;
    }
```

Cycle de vie des objets

Invocation des méthodes d'instances

- invocation de la méthode `travailler` des objets référencés par e1 et e2

```
e1.travailler(2);
e2.travailler(2);
e2.travailler(3);
```



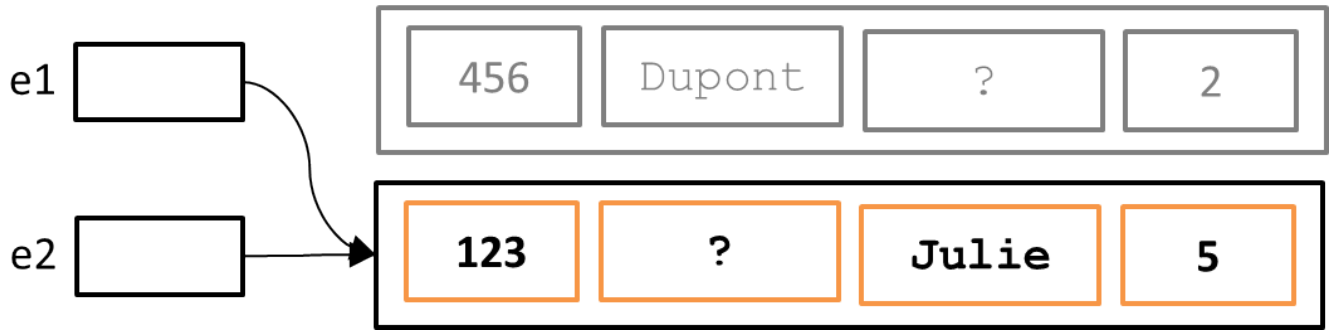

```
class Etudiant {
    // ...
    public void travailler(int n) {
        nbreHeureTravail += n;
    }
    // ...
}
```

Cycle de vie des objets

Affectation d'une référence

- e1 reçoit la référence de e2 : e1 et e2 ont maintenant pour référence la même instance
- l'instance initiale référencée par e1 n'est plus référencée, elle n'est donc plus accessible : elle est perdue et sera éliminée par le ramasse-miettes

```
e1 = e2;
```



Encapsulation et modificateurs d'accès

Accès contrôlé aux classes et aux membres d'une classe
Un modificateur précise l'accessibilité d'une classe et de ses membres :

- au niveau d'une classe : **public** ou, par défaut, limité au package
- au niveau des membres : **public**, **private**, **protected**, ou par défaut, limité au package

Niveau d'accès et visibilité :

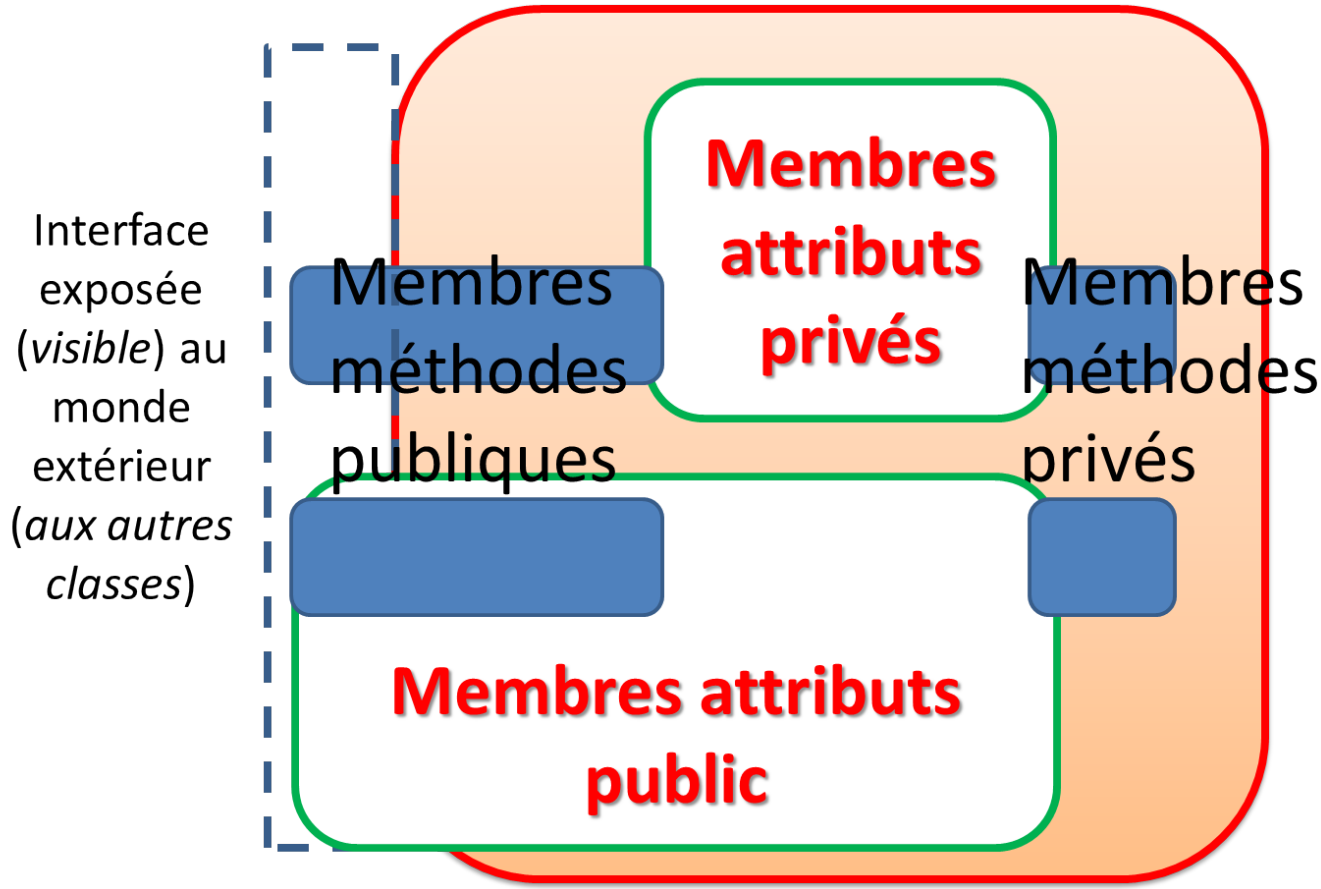
Modificateur	classe	package	sous-classe	tous
public	O	O	O	O
protected	O	O	O	N
(par défaut)	O	O	N	N
private	O	N	N	N

En général, il est conseillé :

- pour les membres attributs d'une classe : **private**
- pour les membres méthodes d'une classe : **public**

Encapsulation et modificateurs d'accès

Encapsulation, exposition de l'interface d'accès aux attributs d'un objet



Les méthodes permettent de fournir sélectivement un accès en lecture ou modification aux membres attributs.

Héritage

L'héritage offre des mécanismes permettant d'utiliser des classes d'objets existantes afin de les spécialiser.

Les nouvelles classes développées bénéficient ainsi de tous les membres visibles des classes dont elles héritent, qu'elles soient des classes métiers ou des classes de l'API Java

La durée et le nombre de lignes de codes produits s'en trouvent réduit, la fiabilité des programmes s'en trouve accrue : réutilisation et extension de classes existantes

Héritage

Héritage, **extends**

- généralisation : une classe mère (super-classe) regroupe les membres communs aux classes spécialisées
- spécialisation (anglais IS-A): des classes filles (sous-classes ou classes dérivées) définissent les membres spécifiques, héritent des membres visibles de leur super-classe et peuvent invoquer le constructeur de leur super-classe : **super(...)**

Une méthode de la super-classe peut être redéfinie/spécialisée dans les classes filles : polymorphisme de redéfinition (même signature)

En Java, l'héritage est simple (*on ne peut hériter que d'une seule classe*)

```
class B extends A { // B herite de A
    // ...
}
```

Les mécanismes d'interfaces permettent cependant de palier à l'absence d'héritage multiple.

Héritage

Une classe générique capitalise les membres communs à tous les animaux :

```
class Animal {
    private String nom;
    private float poids;
    public void emettreUnSon() {
        System.out.println("?");
    }
}
class Chien extends Animal {
    public void emettreUnSon() {
        System.out.println("ouahouah");
    }
}
class Chat extends Animal {
    public void emettreUnSon() {
        System.out.println("miaoumiaou");
    }
}
```

Héritage

Exemple

```
Animal a = new Animal();
Chat b = new Chat();
Chien c = new Chien();
a.emettreUnSon();
b.emettreUnSon();
c.emettreUnSon();

Animal d = new Chat();
d.emettreUnSon();
```

ce qui donne :

?

```
miaoumiaou
ouahouah
miaoumiaou
```

Héritage et polymorphisme

Le polymorphisme de redéfinition (anglais override) : 2 méthodes visibles portent le même nom et possèdent la même signature

```
class Animal {
    // ...
    public void emettreUnSon() {
        System.out.println("?");
    }
}
class Chien extends Animal {
    // ...
    public void emettreUnSon() {
        System.out.println("ouahouah");
    }
}
```

La méthode choisie sera celle correspondant à l'objet courant.

Classes abstraites

Classes dans lesquelles certaines méthodes restent abstraites (non définies)
Les classes abstraites ne peuvent être instanciées, les méthodes abstraites doivent être implémentées par les classes concrètes (les classes filles).
C'est une forme de contrat à respecter par les classes filles.
Une classe abstraite :

- peut définir des constantes
- peut définir des méthodes finales (non modifiables par les enfants)
- peut implémenter des interfaces
- peut servir de classes de base

Classes et membres abstraits

Définir une classe ou certains membres comme **abstract** afin d'obliger les classes spécialisées à le définir :

```
public abstract class Animal {
    private String nom;
    private float poids;
    public abstract void emettreUnSon();
    }
}
class Chien extends Animal {
    public void emettreUnSon() {
        System.out.println("ouahouah");
    }
}
```

De plus, une classe abstraite ne pourra pas être instanciée :

```
error: Animal is abstract;
 Animal a = new Animal();
            ^
```

Interface

classe dont toutes les méthodes sont abstraites et publiques.

- contiennent seulement des prototypes de méthodes
- ne peuvent être instanciés
- peuvent contenir des attributs initialisés
- toutes les méthodes devront être implémentées
- une interface peuvent hériter d'autres interfaces

```
[visibilité] interface NomInterface [extends autres interfaces] {
        déclaration de constantes
        déclaration de méthodes  (signature seule)
}
```

Une classe qui implémente une interface (**implements**) doit en implémenter toutes les méthodes.

Interface

Exemple d'utilisation de **implements** : la classe Etudiant doit absolument définir ce qu'est qu'étudier :

```
public interface ITravailleur {
    public void etudier(int duree);
}
class Etudiant implements ITravailleur {
    private int tempsTravail;
    public void etudier(int duree) {
      tempsTravail+=duree;
    }
}
```

Différences par rapport aux classes abstraites : on peut implémenter plusieurs interfaces, mais n'hériter que d'une seule classe.

Modificateur final

Le modificateur **final** permet la protection contre des modifications.
Il peut s'appliquer à plusieurs niveaux :

- une classe : empêche l'extension de cette classe

```
public final class MaClasse { ... }
```

(**entends** `MaClasse` : interdit)

- un attribut : il ne peut être initialisé qu'une seule fois (= constante)

```
public class MaClasse {
    public final double TAUX = 1.196;
}
```

(nouvelle affectation de valeur à `TAUX` : interdit)

- une méthode : empêche sa redéfinition

```
public class MaClasse {
    public final void maMethode() {...}
}
```

(possible d'étendre `MaClasse`, mais pas redéfinir `maMethode`)

Package

Package (paquetage), mécanisme d'organisation des classes

- groupement des classes et d'interfaces ayant un lien
- assure une protection d'accès
- évite les conflits de nom
- toutes les classes d'un **package** doivent comporter une 1ère ligne avec

  `package <nom du package>;`
- le nom du package décrit une hiérarchie correspondant à une arborescence de stockage des classes compilées
- pour utiliser un package ou une classe d'un package, il faut l'importer : **import**

```
import <nom du package>.UneClasse;
import <nom du package>.*;
```

Package - arborescence

La package correspond au positionnement d'une classe dans une arborescence. Le mot-clef **package** en tout début de code source peut préciser un nom de package pour la classe. Sinon, la classe n'appartient à aucun package (ou package par défaut qui correspond au répertoire où elles se trouvent : à éviter pour des développements professionnels).

Exemple du fichier "Animal.java" :

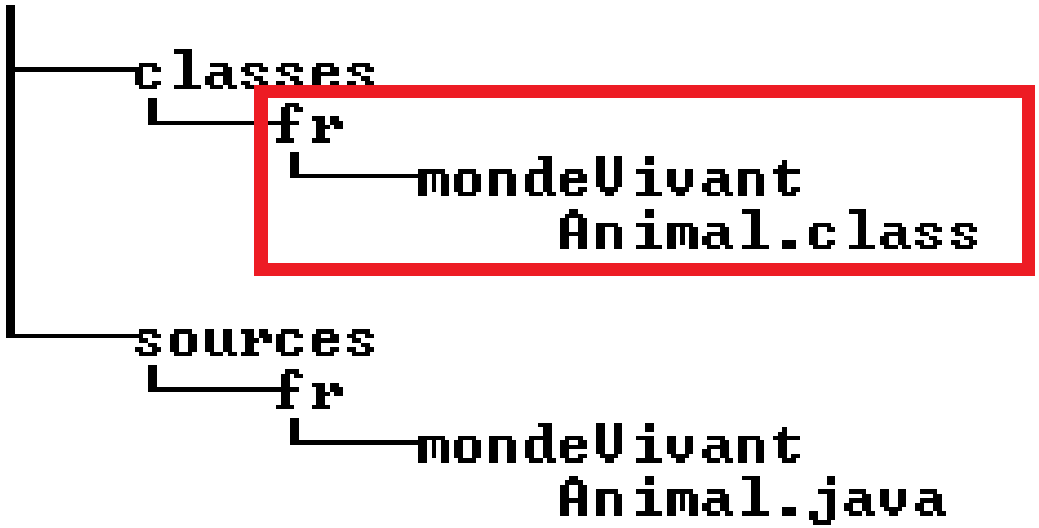
```
package fr.mondeVivant;
class Animal {
    // ...
}
```

La classe compilée (Animal.class) sera localisée dans l'arborescence "fr/mondeVivant/" .

Package - utilisation

Pour utiliser la classe Animal dans une classe qui se trouve dans un autre package : **import**

```
import fr.mondeVivant.*;
class MondeAnimal {
    Animal a;
}
```



Généricité

La généricité est la capacité d'une classe à pouvoir préciser le type de ses membres de manière variable.

Exemple pour une liste de "choses" :

```
public class ListeDe<Chose> {
    private LinkedList<Chose> liste;
    public void ajouter(Chose c) {
        liste.add(c);
    }
}
```

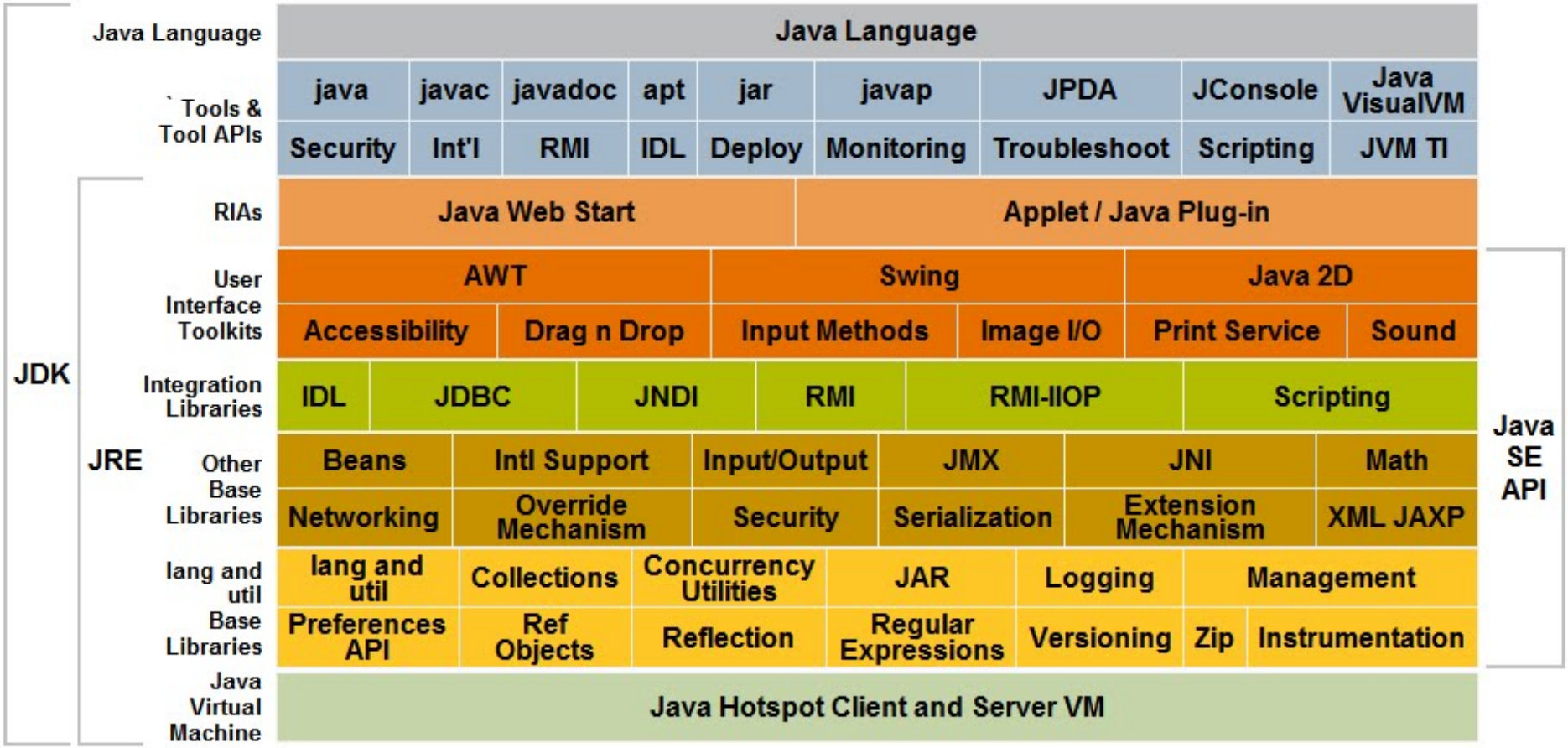
On pourra ainsi définir quel est le type de ces choses :

```
ListeDe<String> maListe1 = new ListeDe<String>();
ListeDe<Animal> maListe2 = new ListeDe<Animal>();
```

Cela permet de capitaliser des mécanismes génériques et de les appliquer à n'importe quel type d'objet.

API Java

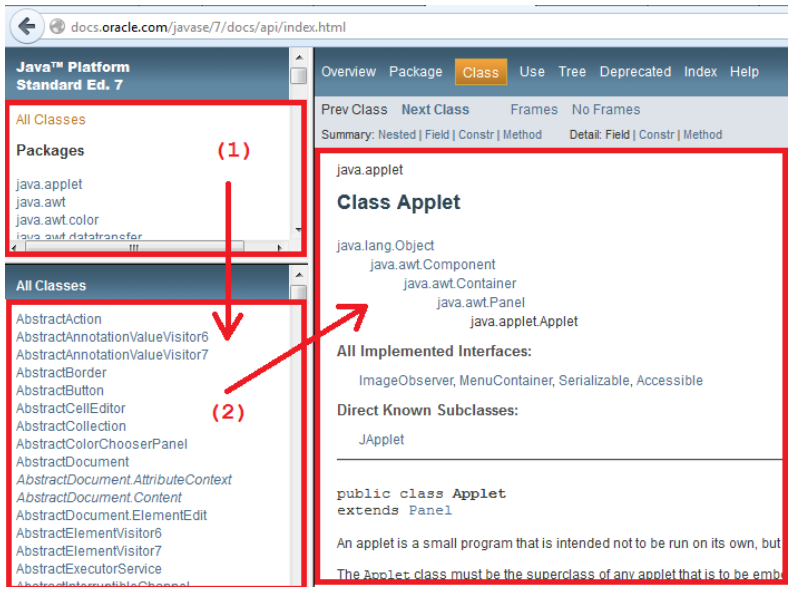
API Java



API Java

- ensemble de classes
- organisées en packages
- permettant de répondre à une vaste étendue de problèmes

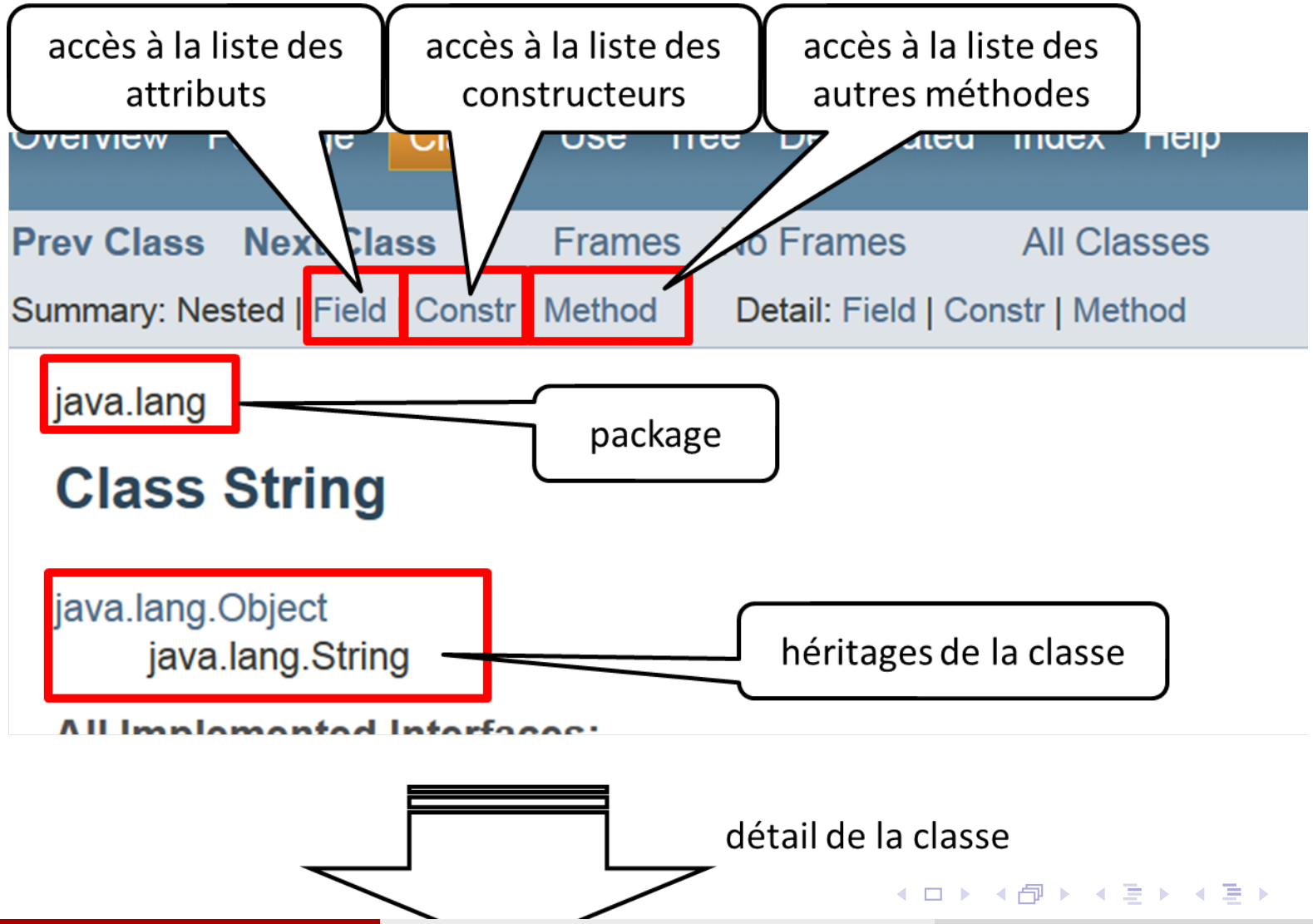
API Java 7



1. sélectionner un package
2. sélectionner une classe ou une interface

API Java

Exemple d'une classe String : API Java 7 - String



API Java

Quelques packages :

- java.lang : classes fondamentales de Java (implicitement importé)
- java.util : services du système d'exploitation
- java.awt : composants graphiques de base
- javax.swing : composants graphiques plus performants
- java.io : gestion des entrées/sorties
- java.net : fonctions réseaux

API Java 7

Classe String

package java.lang

- classe représentant une chaine de caractères
- supporte Unicode
- possède un opérateur de concaténation +
- ne nécessite pas d'instanciation
- sa valeur n'est pas modifiable, mais toute nouvelle affectation crée une nouvelle instance

```
String message = "bonjour !";
String prenom = "Paul";
String messageComplet = message + prenom;
```

Classe String

Quelques méthodes invocables sur un objet de classe String :

- equals(String) : tester l'égalité du contenu de 2 chaines
- substring(int, int) : extraire une sous-chaine à partir d'une position sur une longueur
- toUpperCase() : renvoyer la valeur en lettres majuscules
- toLowerCase() : renvoyer la valeur en lettres minuscules
- length() : renvoyer la longueur de la chaine

Exemples :

```
System.out.println(message.toUpperCase());
System.out.println(prenom.length());
if (prenom.equals("jacques")) ...;
```

Classes de nombres

package java.lang

- classes encapsulant les types de base
- nombreuses méthodes associées
- conversion automatique d'un type primitif quand c'est nécessaire

Classes :

- grands entiers : BigInteger
- grands réel : BigDecimal
- entiers : Byte(byte), Short(short), Integer(int), Long(long)
- réels : Float(float), Double(double)
- booléen : Boolean(boolean)
- caractère : Character(char)

```
Integer i = 1;
Integer i = new Integer(1);
```

Classe System

package java.lang

- classe System déclarée **final** : ne peut être instanciée
- comporte le membre attribut **static out** de la classe `java.io.Printstream` (l'objet *out* est directement utilisable)

Méthodes :

- println(Objet) : envoyer sur le flux puis passer à la ligne
- print(Objet) : envoyer sur le flux (sans passer à la ligne)

C'est ainsi qu'on peut invoquer la méthode `println` de l'objet `System.out` :

```
System.out.println("bonjour !");
```

Classe Object

package java.lang

- la classe Object est la mère de toutes les classes (*toutes les classes en héritent implicitement*)
- toutes ses méthodes visibles sont donc, par héritage, disponibles à toutes les classes

Méthodes :

- toString() : renvoie une représentation d'un objet sous forme textuelle (*à redéfinir spécifiquement dans chaque classe*)

Exemple de redéfinition dans la classe Animal :

```
public String toString() {
  return "Animal "+nom;
}
```

On pourra ainsi obtenir la "valeur" affichable d'un objet :

```
// ... par exemple dans la classe main :
        Chien a = new Chien("totor");
        System.out.println(a); // affiche "Animal totor"
```

Classe Scanner

classe de java.util

- classe **Scanner** est un lecteur de flux de types primitifs
- un objet doit être instancié avant utilisation
- il doit être associé à un flux : `System.in` est l'objet permettant d'accéder à l'entrée standard du système (le clavier)

```
Scanner sc = new Scanner(System.in);
```

Méthodes :

- nextInt() : lit le prochain entier
- nextFloat() : lit le prochain réel
- nextLine() : lit la prochaine ligne
- hasNext() : indique s'il y a quelque chose à lire

```
int i;
i = sc.nextInt();
```

Classe Math

package java.lang, **final**, disposant d'attributs et de méthodes **static**

- PI : valeur de PI

```
circonf = 2 * Math.PI * rayon;
```

Méthodes static :

- abs(double) : valeur absolue
- cos(double) : cosinus d'un angle exprimé en radians
- pow(double,double) : elever un nombre à la puissance d'un autre nombre
- random() : retourne un nombre pseudo-aléatoire dans la plage $[0.0, 1.0[$
- round(double) : arrondit
- toRadians(double) : convertit de degrés en radians

```
int i;
i = Math.round(2.56);
```

Classe Array

classe de java.util

- classe représentant un tableau
- sa capacité est fixe
- il peut contenir des objets d'un seul type

Méthodes

- sort(int[]) : trier les nombre entiers d'un tableau
- sort(Object[]) : trier les objets d'un tableau
- binarySearch(int) : recherche dans un tableau trié
- indexOf(Object) : position d'un objet dans la liste
- equals(int[],int[]) : test l'égalité entre 2 tableaux
- fill(int[],int) : remplit chaque élément avec la même valeur

Classe Vector et ArrayList

classe de java.util

- classe représentant un tableau
- capacité variable
- peut contenir des objets de différents types

Méthodes :

- add(Object) : ajouter un objet à la liste
- add(int,Object) : ajouter un objet à une certaine position dans la liste
- get(int) : obtenir un objet à un indice
- indexOf(Object) : position d'un objet dans la liste
- remove(int) : enlever l'objet à la position
- size() : nombre d'éléments de la liste
- isEmpty() : indique si la liste est vide ou non

La différence essentielle : **Vector** gère la sécurité d'accès

Interfaces graphiques et évènements

- exécutée sur le poste client (client "lourd")
- nécessite la présence de la VM Java
- API AWT(+portable, +rapide, appel natifs)
- API Swing (+récent, +riche, +lent)
- Swing s'appuie sur AWT (même gestionnaire d'évènements)
- Swing : composants légers(ne s'appuient pas sur le S.E), permettent de définir un "look and feel" spécifique

Interfaces graphiques et évènements

Java Foundation Classes : ensemble d'éléments permettant la construction d'interfaces graphique utilisateur (GUI) riches

- composents graphique Swing
- look-and-feel paramétrable
- technologies de lecture d'écran et affichage Braille
- API Java 2d
- internationalisation

Interfaces graphiques et évènements

La gestion d'interface graphique utilisant le paquetage Swing fait intervenir les composants suivants :

- composants graphiques : JLabel, JButton, JCheckBox, JRadioButton, JList, JTree
- conteneurs primaires : JFrame, JApplet, JDialog
- conteneurs secondaires : JPanel, JScrollPane, JSplitPane, JTabbedPane
- gestionnaires de placement : aucun, BorderLayout, FlowLayout
- gestionnaires d'évènements

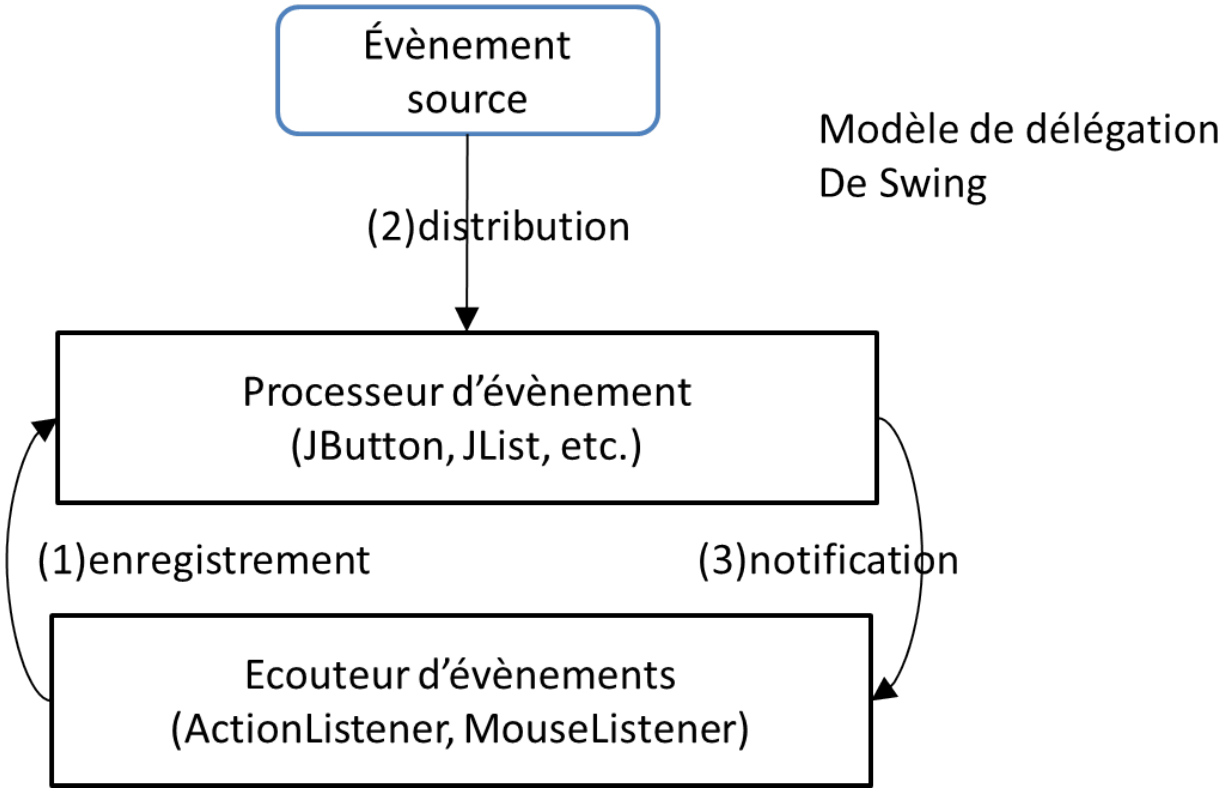
Interfaces graphiques et évènements - API

Exemple de fenêtre :

```
import javax.swing.*;
public class Fenetre1 extends JFrame {
  JButton btn;
  public Fenetre1() {
    btn = new JButton ("cliquer ici !");
    add(btn);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle("ma fenetre");
    pack();
    setVisible(true);
  }
  public static void main(String argv[ ]) {
    Fenetre1 fen = new Fenetre1();
  }
}
```

Interfaces graphiques et évènements

Les évènements sont des objets. Les composants graphiques implémentent les interfaces d'écouteurs d'évènement (Listener)



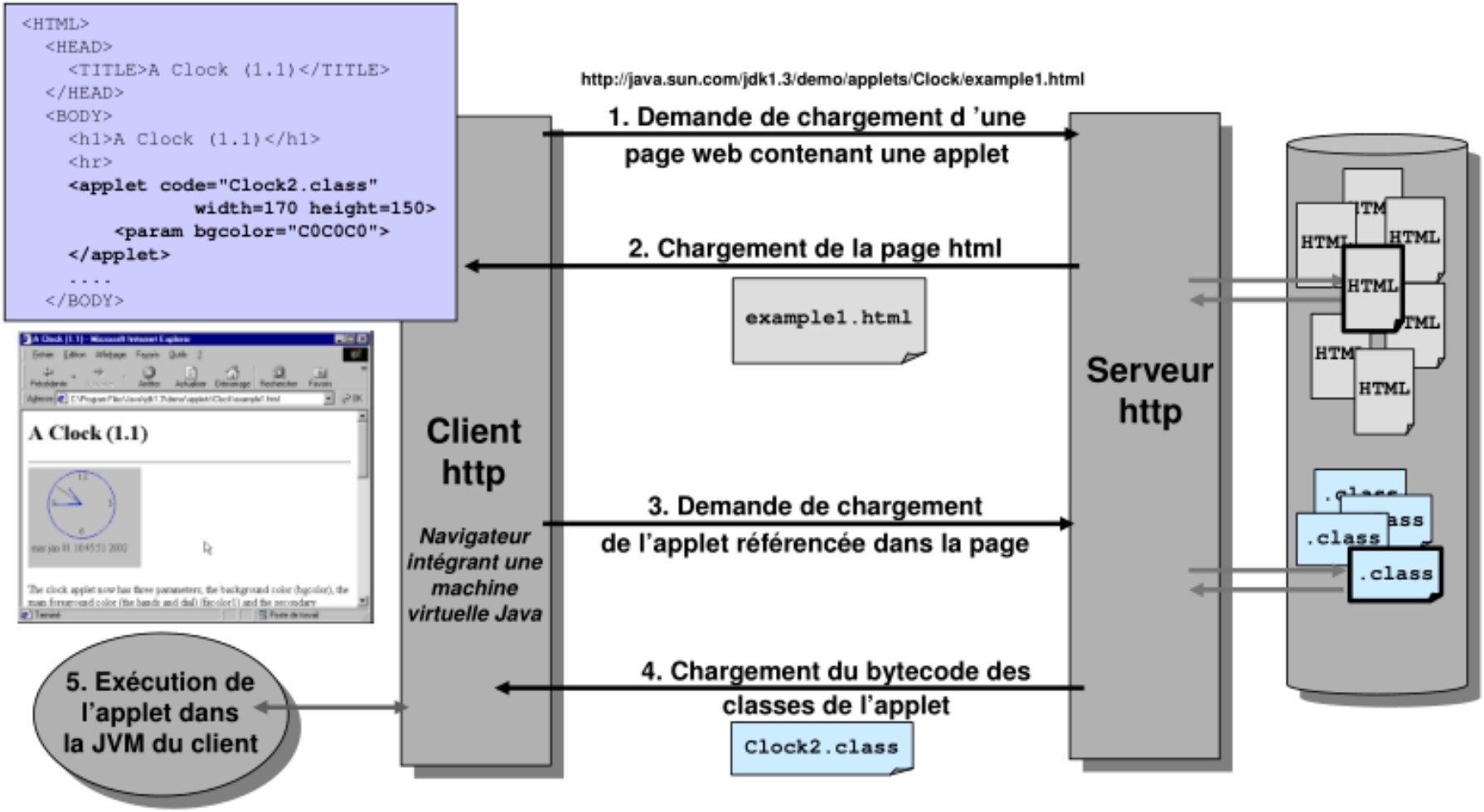
Interfaces graphiques et évènements - API

Exemple de fenêtre :

```
import javax.swing.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
public class Fenetre2 extends JFrame implements
    ActionListener {
  JButton btn;
  public Fenetre2() {
    btn = new JButton ("cliquer ici !");
    add(btn);
    btn.addActionListener(this);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setTitle("ma fenetre");
    pack(); setVisible(true);
  }
  public void actionPerformed(ActionEvent e) {
    btn.setText("ok"); }
  public static void main(String argv[ ]) {
    Fenetre2 fen = new Fenetre2();
```

Applets

Code Java mobile exécuté généralement dans un navigateur (page web)



Classe JApplet : extension de Applet pour utiliser JFC

Applets

- code mobile
- appel intégré à une page Web
- exécuté sur le poste client (client "léger")
- nécessite la présence de la JVM sur le poste client

```
import java.applet.*;
public class MonApplet extends Applet {
    public void init() {
    }
    public void start() {
    }
    public void stop() {
    }
    public void destroy() {
    }
}
```

Applets

Exemple :

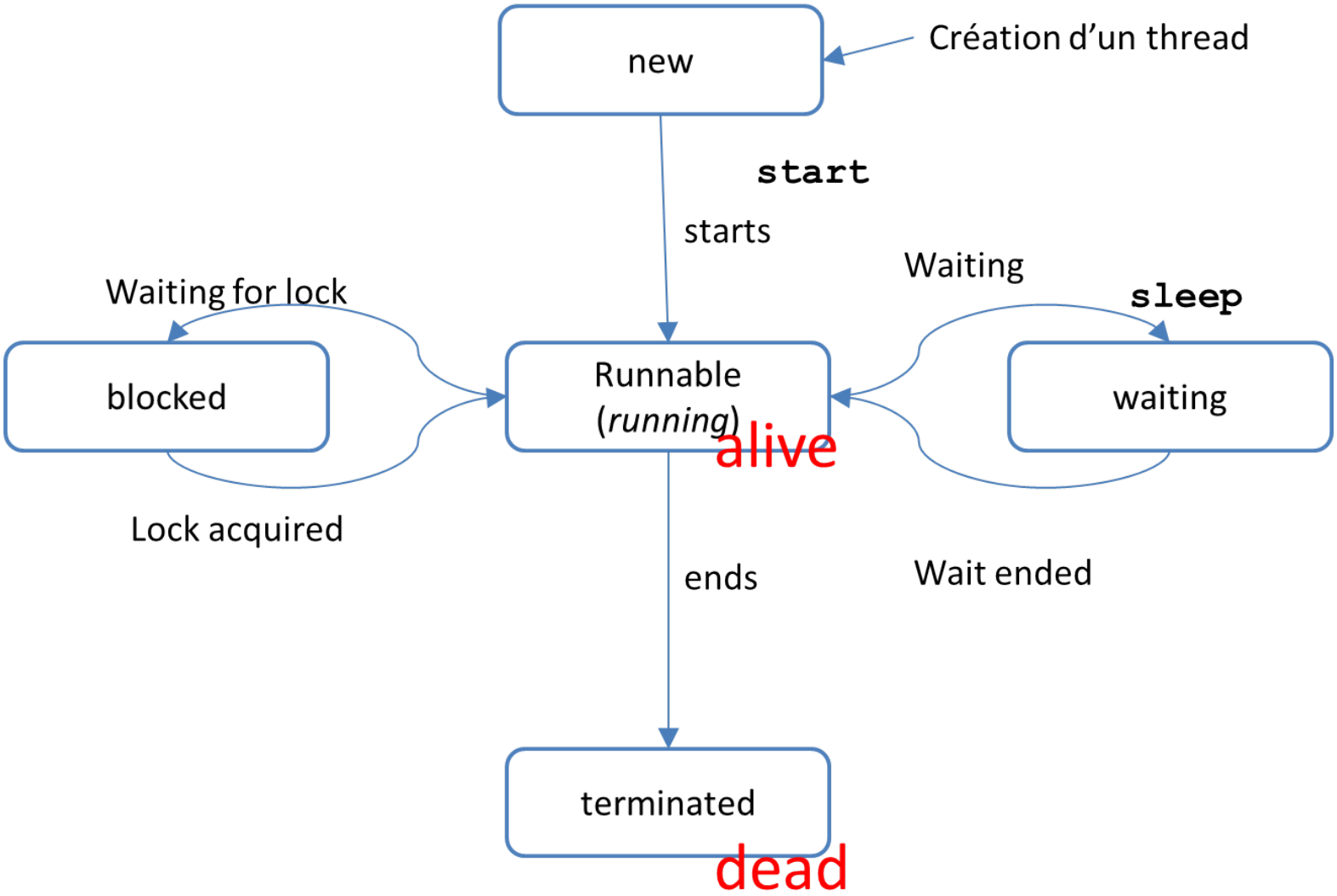
```
import java.applet.*;
import java.awt.*;
public class Applet1 extends Applet {
  private String message;
  public void init() {
message = "hello !";
  }
  public void stop() { }
  public void paint(Graphics g) {
    g.drawString(message, 20,10);
  }
}
```

Exécution parallèle : multithreading

- multitraitement java : exécution concurrente de portions de code
- optimisation de l'utilisation du CPU
- exécuté au sein de l'espace mémoire d'un processus
- verrouillage et libération de ressources (**synchronized**)

Threads

Les états d'un thread



Thread

Exemple d'une classe spécialisant la classe Thread :

```
public class Thread1  extends Thread {
  private String str;
  public Thread(String str) {
    this.str = str;
  }
  public void run() {
    long debut = System.currentTimeMillis();
    while( System.currentTimeMillis() < ( debut + (1000 *
        10))) {
      System.out.println("thread "+str);
      try {
        sleep(500);
      }
      catch (InterruptedException ex) {}
    }
  }
```

L'exécution du thread dure 10 secondes.

Thread

Exemple :

```
  public static void main(String argv[ ]) {
    Thread1 t1 = new Thread1("t1") ;
    t1.start();
    Thread1 t2 = new Thread1("t2");
    t2.start();
    while( t2.isAlive() ) {
      System.out.println("Main");
      try {
        sleep(800);
      }
      catch (InterruptedException ex) {}
    }
  }
}
```

Deux threads sont instanciés : la procédure principale attend que le 2nd ne soit plus vivant.

Réseau

- abstraction des couches matérielles
- offre des accès aux serveurs distants
- permet la réalisation de clients et serveurs
- classes : InetAddress, URL, Socket et Datagramme

```
import java.net.*;
class InetAddressTest
{
  public static void main(String args[]) throws
      UnknownHostException {
    InetAddress adr = InetAddress.getLocalHost();
    System.out.println(adr);
    adr = InetAddress.getByName("www.univ-littoral.fr");
    System.out.println(adr);
  }

}
```