Java Avance

Eléments de programmation

Emmanuel ADAM

Université Polytechnique des Hauts-De-France

UPHF/INSA HdF

1 Présentation de Java

2 Eléments de programmation
- Variables et fonctions
- Primitives
- Tableaux
- Opérateurs
- Branchements conditionnels
- Sortie de bloc

3 Concepts objets
- Eléments de classes
- Références
- Référence à soi-même
- Modificateur d'accès
- Héritage
- Généricité
- Comparaison
- Membres de classe (statiques)

4 Concept Objet avancé

- Classes abstraites
- Interface
- Énumération
- Record
- Classe interne
- Classe interne
- Classe anonyme
- Les paquetages

Petite histoire

Evolution permanente des versions

1996, Java (1.0)
1997, V 1.1 : JDBC (bases de données), Composants, RMI (réseau)
1998, V 1.2 : Swing (fenêtres graphiques)
2002, V 1.4 : assert, expressions régulières, parseur XML, exceptions chaînées
2004, V 1.5 (J2SE 5.0) : autoboxing, for étendu, types génériques, nb variables arguments
2009, reprise par Oracle ; mises à jours régulières
2014, Java 8 : ajout de JavaFX, **Programmation fonctionnelle**
2018, Java 10 : déclaration de variables non typées
2019, Java 11 : retrait de JavaFX ! Amélioration du *Garbage Collector*
2019, Java 12 : gestion de mémoire augmentée
2020, Java 15 : notion de *record* (structure), gestion de *switch*, ...

La Compilation & JAVA

Particularité : ByteCode

- Dans les 3 langages de programmations les plus utilisés depuis des années
- Le principal atout de JAVA concerne sa portabilité.
- *Write Once, Run Anywhere (WORA)*
- Celle-ci est possible par la notion de **ByteCode**.
- JAVA ne produit pas un code natif, directement exécutable.
- JAVA produit un code intermédiaire, le ByteCode, interprétable par une machine virtuelle java (**JVM**).
- Le **même** bytecode s'exécute donc sur toute machine disposant d'une JVM

Bytecode vs Code natif

Propriétés du bytecode

- (+) source portable
- (+) bytecode portable
- (-) interprété donc moins rapide (temps de chargement)

Propriétés du code natif

- (+) rapidité d'exécution (code adapté à la machine)
- (-) source pas toujours portable
- (-) recompiler pour s'adapter aux systèmes d'exploitation/machines

Exemple de compilation

Source Java

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

ByteCode

Génération du ByteCode par le compilateur javac →
`javac Hello.java`

Exécution via la machine virtuelle Java (JVM) →
`java Hello`
→ Salut!!

Gestion de la mémoire

Tout est objet

- En JAVA, sauf types primitifs (entiers, réels, ...), tout est objet.
- A chaque objet créé, une zone mémoire est dynamiquement allouée
- **En Java, ne peut détruire explicitement les objets**
- Dès qu'un objet n'est plus utilisé (sortie de boucle, ...), il peut être détruit.
- Le **Garbage Collector** (ramasse-miettes) se charge de récupérer la mémoire.

Ramasse-Miettes

Ramasse-Miettes : Garbage collector

- Fuites graves de mémoire (hard leaks) si inexistence de système de nettoyage
  - Géré par le GC (Garbage Collector)
  - surveillance de l'utilisation des objets,
  - détermination des objets devenus inutiles,
  - récupération des ressources libérées.
- Fuites légères (soft leaks) : un programme encombre inutilement la mémoire
  - A gérer par le développeur

Fonctionnement du Ramasse-Miettes

Garbage collector

- La Machine Virtuelle Java (`JVM`) surveille le nombre *nb* de **références** à chaque objet.
- Si $nb = 0 \rightarrow$ objet inutilisé $\rightarrow$ libérer mémoire
- Si $nb \neq 0 \rightarrow$ objet utilisé $\rightarrow$ incrémenter âge objet
  - Si $age \leq seuil \rightarrow$ objet jeune $\rightarrow$ placer.laisser dans pile des jeunes objets (*YoungGen*) traités régulièrement
  - Si $age > seuil \rightarrow$ objet pérenne $\rightarrow$ ranger dans pile des vieux objets (*OldGen*), traités plus rarement

Variables et fonctions

Déclarations

- Variables
  - `type` *nomVariable* = *valeur*
  - depuis java 10, possibilité de ne pas préciser le type :
    `var` *nomVariable* = *valeur*
    le type est alors fixé à la compilation
- Fonction
  - `type` *nomFonction*($\{type_i\ nomVariable_i, \}^*$)
  - le retour de valeur en fin de fonction est réalisé par :
    `return` *valeur*;
  - une procédure ne retournant rien est de type `void`

Tout est Objet (ou presque...)

java.lang.Object

Types primitifs

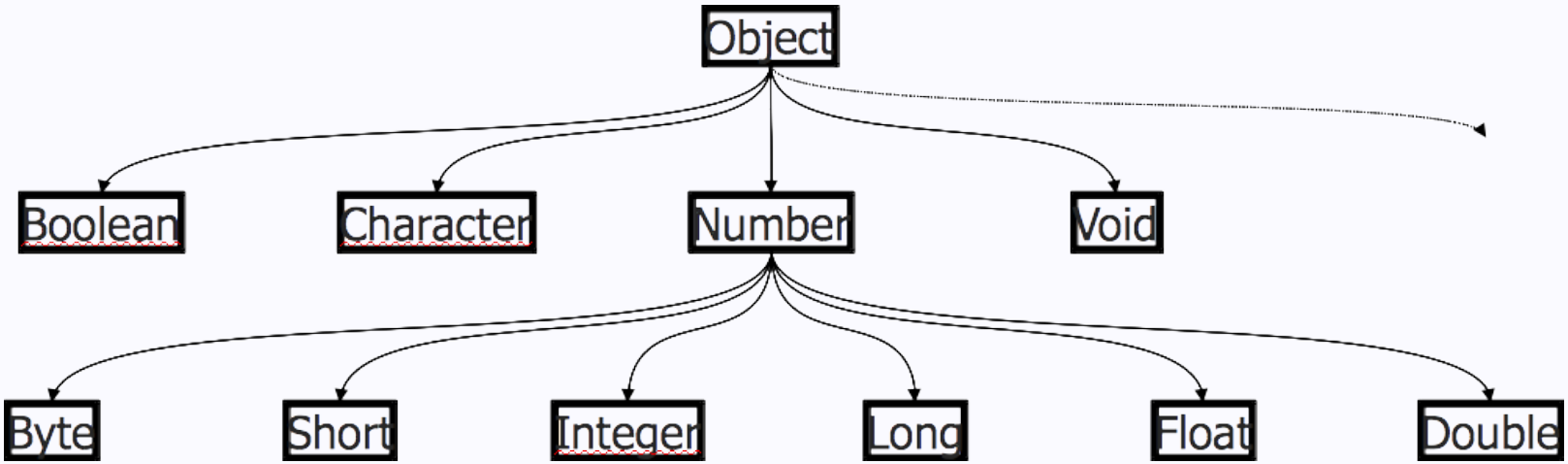
- Tous les éléments en JAVA dérivent de la classe **Object**.
- Tous sauf les **primitives** représentant des types simples.
- A chaque type de primitive correspond une classe d'objet.

Primitive	*nb bits*	*mini*	*maxi*	**Classe**
boolean	1	false	true	Boolean
char	16	Unicode 0	Unicode 216-1	Character
byte	8	-128	127	Byte
short	16	-2^{15}	$2^{15}-1$	Short
int	32	-2^{31}	$2^{31}-1$	Integer
long	64	-2^{63}	$2^{63}-1$	Long
float	32	$(+/-)1.4\ 10^{-45}$	$(+/-)3.4\ 10^{38}$	Float
double	64	$(+/-)4.9\ 10^{-324}$	$(+/-)1.8\ 10^{308}$	Double
void	.	.	.	Void

Tout est Objet (ou presque !)

Objets et Primitives

- A chaque type de primitive correspond une classe d'objet permettant les conversions.



Conversion Primitives - Objets

Conversion explicite ou implicite

- **Conversion primitive → objet**
  - Supposons `int i=5;`
  - Conversion explicite : `Integer obj = Integer.valueOf(i);`
  - Conversion implicite : `Integer obj = i;`
- **Conversion objet → primitive**
  - Supposons `Double obj = Double.valueOf(3.14);`
  - Conversion explicite : `double d = obj.doubleValue();`
  - Conversion implicite : `double d = obj;`
- Inconvénient de l'utilisation d'objets : le coût spatial et temporel !
- Avantages de l'utilisation d'objets : les méthodes de conversion `intValue(); doubleValue(); toString(); ...`
- Remarque pour réels simple (ajouter un 'f') : `float f = 6.5f;`

Variables depuis Java 10

Inférence de type

- Depuis Java 10, la déclaration de variables locales est simplifiée
- Par défaut, une valeur numérique est une primitive
  - `var` $i = 5$;
    implique que i est de type `int`
  - `var` $oi = new\ Integer(5)$;
    implique que oi est de type `Integer`
- L'inférence nécessite une valeur d'un type identifiable
  - `var liste = null;`
    est impossible !
  - `var liste = new ArrayList<Integer>();`
    est correct
  - `ArrayList<Integer> liste = null;`
    utilisable avant java 10 est aussi correct

Les tableaux

Tableau = objet à unique attribut

- Un tableau en Java est un type d'objet simple (il ne contient que l'attribut `length`).
- Tableau simple : `int tab [] = new int [10];` tableau de 10 entiers de tab[0] à tab[9]. (tab.length=10)
- Tableau multi-dimensionnel :
  `double cube[][][] = new double[3][3][3];`
- Initialisation : `int tab[] = {1, 2, 3, 4, 5};`
- Création et passage de tableau en argument :
  `afficheTab(new int[]{1,2,3});`

Les opérateurs

Opérateurs arithmétiques

pour tout nombre x, on peut écrire : `x++; x-; ++x; -x; x+=2; x-=2; x*=2; x/=2; x=5%2 (x = 5 modulo 2)`

Opérateurs logiques

pour tout booléen a et b :

- **non** a ≡ `(!a)`
- a **et** b ≡ `(a & b)` ou de préférence `(a && b)` (test économe)
- a **ou** b ≡ `(a | b)` ou de préférence `(a || b)` (test économe)
- a **ou exclusif** b ≡ `(a ^ b)`

Opérateur ternaire

`(test?valOk:valSinon)` : si *test* réussi alors retourner *valOk* sinon retourner *valSinon*

`x = (a>b?5:-10)` : si $a > b$ alors x reçoit 5 sinon reçoit -10

Les Branchements conditionnels (1/2)

Conditions : Si ...Alors ...Sinon

```
if (test)
  {
   action1();
   action2();
  }
else
  {
   action1();
   action3();
   action4();
  }
```

Les Branchements conditionnels (2/3)

Choix multiple (version <12)

```
switch (noJour)
{
   case 1 :
   case 2 :
   case 3 :
   case 4 :
   case 5 : System.out.println("Semaine"); break;
   case 6 : System.out.println("Samedi"); break;
   case 7 : System.out.println("Dimanche"); break;
   default : System.out.println("no de jour inconnu !");;
}
```

Les Branchements conditionnels (2/3)

Choix multiple (possible depuis version 12)

```
switch (noJour)
{
   case 1,2,3,4,5 -> System.out.println("Semaine");
   case 6 -> System.out.println("Samedi");
   case 7 -> System.out.println("Dimanche");
   default -> System.out.println("Jour inconnu");
}
```

Autre choix multiple (possible depuis version 12)

```
String terme =  switch (noJour) {
   case 1,2,3,4,5 -> "Semaine";
   case 6 -> "Samedi";
   case 7 -> "Dimanche";
   default -> "Jour inconnu";
};
```

Les Boucles (1/2)

Boucle "Tant que"

```
while (test)
{
   actions();
}
...
do
{
   actions();
}
while (test);
```

Les Boucles (2/2)

Boucle "Pour"

```
int i;
for(i=0; i<5; i++) // ou for(int i=0; i<5; i++)
{
    actions();
}
```

```
int add(int ... tab)
{
    int som=0;
    for (int val : tab)
        som += val;
    return som;
}
```

```
int res = add(15,22,13,64);
```

Les sorties de blocs

Passer des lignes

- L'instruction `break` permet de sortir d'une boucle avant sa fin "normale".
- L'instruction `continue` passe le reste de la boucle, mais n'en sort pas.

Concept de classe et d'objet

Classe et objets

- Une classe est composée de membres
  - des **attributs** (champs), définissant l'état d'un objet,
  - des **méthodes** (fonctions, procédures) agissant sur ou en fonction des attributs,
  - un ou des **constructeurs**, initialisant l'objet.
- Tout objet
  - instancie une classe
  - occupe un emplacement dans la mémoire de la JVM
- En java, l'adresse d'un objet est appelé référence
- Les variables portent les références aux objets
- Une variable typée de valeur null ne pointe sur aucune référence

Exemple de classe

```java
/**classe simple representant  une personne */
public class Personne {
    //   les attributs
    /**nom de la personne*/
    String nom;
    /**age de la personne*/
    int age;

    //   les constructeurs
    /**constructeur par defaut*/
    Personne(){nom=""; age=18;}
    /**constructeur complet*/
    Personne(String _nom, int _age){nom=_nom; age=_age;}
    /**constructeur par recopie*/
    Personne(Personne p){nom=p.nom; age=p.age;}

    //   les fonctions
    /**affiche les details sur la personne*/
    void affiche()
    {
      System.out.println("personne , nom="+nom+" , age="+age);
    }
}
```

Exemple d'utilisation d'une classe

```
public class TestPersonne {
    public static void main(String[] args)
    {
        Personne p1 = new Personne();
        Personne p2 = new Personne("adam", 47);
        Personne p3 = new Personne("adam", 47);
        System.out.println(p1);
        p1.affiche();
        System.out.println(p2);
        p2.affiche();
        System.out.println(p3);
        p3.affiche();
    }
}
```

donne :

```
p1=Personne@77459877
personne, nom=, age=18
p2=Personne@eed1f14
personne, nom=adam, age=47
p3=Personne@7229724f
personne, nom=adam, age=47
```

Autre exemple du principe des références (1/3)

```
Personne p1 = new Personne("adam", 47);
//-> creation d'une instance de personne, p1 pointe vers
    cette instance
Personne p2 = p1;
//-> p2 pointe maintenant aussi vers l'instance pointee par
    p1
Personne p3 = new Personne(p1);
//-> creation d'une instance de personne ayant les memes
    caracteristiques que celle pointee par p1, p3 pointe
    vers cette instance
System.out.println("p1="+p1+", p2="+p2+", p3="+p3);
//-> verification des adresses pointees par p1, p2 et p3
```

→ donne :

```
p1=Personne@3b07d329, p2=Personne@3b07d329, p3=Personne@41629346
```

Autre exemple du principe des références (2/3)

```
System.out.println("caracteristiques de p1, p2 et p3");
p1.affiche();
p2.affiche();
p3.affiche();
```

→ donne :

```
caracteristiques de p1, p2 et p3
personne, nom=adam, age=47
personne, nom=adam, age=47
personne, nom=adam, age=47
```

Autre exemple du principe des références (3/3)

```
//changement du nom de la personne pointee par p2 et p1
p2.nom = "eve";
//verification de l'impact de la modification
System.out.println("caracteristiques de p1, p2 et p3");
p1.affiche();
p2.affiche();
//p3 normalement n'a pas change..
p3.affiche();
```

→ donne :

```
caracteristiques de p1, p2 et p3
personne, nom=eve, age=47
personne, nom=eve, age=47
personne, nom=adam, age=47
```

Tableau d'objets

Stocker des objets dans un tableau

- Un tableau d'objets est un tableau de références à des objets
- Il faut définir la taille du tableau
- Puis, placer dans chaque case une référence à un objet créé

```
Personne [] tableau = new Personne[5];
for (int i=0; i<5; i++)
  tableau[i] = new Personne ("nom"+i, 20+i);
for (Personne p:tableau) p.affiche();
```

→ donne :

```
personne, nom=nom0, age=20
personne, nom=nom1, age=21
personne, nom=nom2, age=22
personne, nom=nom3, age=23
personne, nom=nom4, age=24
```

Référence à soi-même

Se connaître

- En Java, une instance d'objet peut accéder à sa référence (son adresse) par le mot clé **this**
- **this** peut être utilisé pour différencier ses attributs des variables locales

```
Personne(String nom, int age){
    this.nom = nom; this.age = age;}
```

- **this** peut être utilisé pour se passer en paramètre

```
void chercherAmi(Personne autre){
  if(autre.age>=18 && autre.age<=25)
    autre.demandeAjoutAmi(this);
}
```

Modificateur d'accès

Gérer l'accès aux membres

- Une classe peut protéger ou ouvrir l'accès à ses membres par des mots clés
  - private : interdit l'accès direct au membre par les objets des autres classes
  - public : permet l'accès au membre par les objets des autres classes
  - protected : permet l'accès au membre par les objets des classes voisines (dans le même répertoires) et descendantes

```
public class Personne {
  // toute instance peut acceder a nom
  public String nom;
  // seules les instances de classes filles et voisines
       peuvent acceder a age
  protected int age;
  // seules les instances de Personne peuvent acceder a
       noSecu
  private int noSecu;
  //...
}
```

Accesseurs, Mutateurs

Acces aux membres privés

- Par défaut, les membres sont de type `protected`
- Pour sécuriser il est préférable de rendre les membres privés et de permettre les accès/modifications par des getters/setters
- pour un attribut `Type nom`,
  - l'accesseur en lecture sera `Type getNom()` : une fonction qui retourne la valeur de l'attribut
  - l'accesseur en écriture sera `void setNom(Type nom)` : une procédure qui affecte à `this.nom` la valeur `nom`

```
public class Personne {
 private int noSecu;
 //...
 public int getNoSecu() { return noSecu; }
 public void setNoSecu(int noSecu){this.noSecu=noSecu;}
}
```

Héritage

Hériter de propriétés, comportements

- En java, si une classe *A* hérite d'une classe *B*,
  elle possède tous les attributs et méthodes de B *non privés*
- Notation :

```
class Personne {...}
class Etudiant extends Personne {}
```

- Il n'y a pas d'héritage multiple en Java

Surcharge (1/3)

Modifier un comportement hérité

- Une classe fille peut surcharger une méthode *non finale* d'une classe mère
- L'objectif de l'héritage est la **généricité** :
  - appliquer le même schéma d'actions à des objets de type différents
  - limiter le nb de noms de méthodes ayant le même objectif

Surcharge (2/3)

```
public class Etudiant extends Personne{
    /**nom de la formation*/
    String formation;

    Etudiant(){formation="?";}
    /**Etudiant herite de Personne
    et donc possede implicitement nom et age*/
    Etudiant(String _nom, int _age, String _formation)
    {nom = _nom; age = _age; formation = _formation;}

    /**surcharge de la methode affiche*/
    void affiche() {
        System.out.println("Etudiant : nom="+nom+" , age="+
            age+" , formation="+formation); }
}
```

Surcharge (3/3)

Empêcher un héritage, une surcharge

- le mot clé `final` permet de bloquer un héritage/une surcharge
- **final** devant la déclaration d'une classe interdit toute autre classe d'hériter de celle-ci
- **final** devant la déclaration d'un attribut le définit en tant que constante *(sa valeur est donnée à la déclaration ou dans un constructeur uniquement)*
- **final** devant la déclaration d'une méthode empêche la surcharge par une classe descendante

Exemple d'applications de la surcharge

```
Personne p1  = new Personne("alpha", 21);
Personne p2  = new Personne("beta", 22);

Etudiant e1  = new Etudiant("gamma", 23, "Java");
Etudiant e2  = new Etudiant("delta", 24, "Java");

//Possibilitee de placer des references a des Etudiant(s)
    dans un tableau de references a des Personne(s)
//car les Etudiant(s) sont des Personne(s)
Personne[] tab = new Personne[]{p1, e1, p2, e2};

//balayage du tableau generique
for(Personne p:tab) p.affiche();
```

→ donne :

```
personne, nom=alpha, age=21
Etudiant : nom=gamma, age=23, formation=Java
personne, nom=beta, age=22
Etudiant : nom=delta, age=24, formation=Java
```

Appel au constructeur parent

Constructeurs implicites/explicites

- Si la classe A hérite de la classe B, le constructeur de A fait **implicitement** appel au constructeur par défaut de B.
- `super(...)` permet de choisir **explicitement** un constructeur parent
- `this(...)` permet de faire appel à un de ses constructeurs

```
class B
{
  int i;
  B(){this.i=10;}
  B(int i){this.i = i;}
}
```

```
class A extends B
{
  int j;
  int k;
  A(){ //implicite appel a B()
  j=20;} //en sortie, i=10 & j=20
  A(int i, int j){super(i); this.j=j;}
  A(int i, int j, int k){
    this(i,j); this.k=k;}
}
```

Appel à une méthode parente

Appel explicite à une méthode originale

- Si la classe A surcharge une méthode de la classe B, elle peut rappeler la méthode parente par le mot clé **super**

```
class Personne
{
  //.....
  void affiche() {
    System.out.println("Personne: nom="+nom+", age="+age); }
}
class Etudiant
{
  //.....
  void affiche() {
    System.out.println("Etudiant: nom="+nom+", age=" + age+"
        , formation="+formation); }
  void afficheLeger() {
    super.affiche(); }
}
```

Généricité (1/2)

Casting

- Les classes Java héritent implicitement de la classe `Object`
- Possibilité de tout stocker dans une liste/un tableau d'`Object`
- $\rightarrow$ utiliser le "casting" pour retrouver les types initiaux
  - (*a* `instanceof` *A*) retourne vrai si *a* est une instance directe ou non de la classe *A*
  - (*a*`.getClass()`==`A.class`) retourne vrai si *a* est une instance *directe* de *A*
  - `instanceof` est plus rapide que le test de classe

Généricité (2/2)

```
Etudiant e1 = new Etudiant("gamma", 23, "Java");
if (e1 instanceof Personne)
  System.out.println("e1 est une instance de la classe
      Personne");
if (e1 instanceof Etudiant)
  System.out.println("e1 est une instance de la classe
      Etudiant");

Personne p1 = new Personne("alpha", 21);
if (p1 instanceof Personne)
  System.out.println("p1 est une instance de la classe
      Personne");
if (p1 instanceof Etudiant)
  System.out.println("p1 est une instance de la classe
      Etudiant");
```

→ donne :

```
e1 est une instance de la classe Personne
e1 est une instance de la classe Etudiant
p1 est une instance de la classe Personne
```

Egalité entre objets (1/2)

Relation binaire d'égalité

- Le test `==` ne compare que les valeurs des références

```
var p1 = new Personne("alpha", 20);
var p2 = new Personne("alpha", 20);
System.out.println(p1==p2);
```

→ *false*

- Java propose de surcharger la fonction
  `public boolean equals(Object obj)`
  qui, pour toute variables x,y,z non nulles, est
  - réflexive : `x.equals(x) == true`
  - symétrique : `x.equals(y) == y.equals(x)`
  - transitive :
    `x.equals(y) && y.equals(z)` ⇒ `x.equals(z) == true`
  - consistante : `x.equals(y)` retourne toujours la même valeur si x et y ne varient pas
- de plus, pour tout x non nul, `x.equals(null) == false`

Egalité entre objets (2/2)

```java
//Dans la classe Personne
/**on decide que deux personnes sont egales
si elles ont le meme nom et meme age*/
@Override
public boolean equals(Object o) {
  //si l'autre porte la meme reference que soi,repondre vrai
  if (this == o) return true;
  // si l'autre est null ou n'est pas de la meme classe,
      repondre faux
  if (o == null || getClass() != o.getClass()) return false;
  // sinon, caster l'autre en tant que Personne
  Personne personne = (Personne) o;
  // tester l'age
  if (age != personne.age) return false;
  // tester le nom
  return (nom == null ? personne.nom == null:
     nom.equals(personne.nom));
  //plus simple, utiliser la classe Objects :
  //return Objects.equals(nom, personne.nom);
}
```

→ `System.out.println(p1.equals(p2));` retourne `true`

Membres de classe (statiques) (1/3)

Partager un membre entre objets

- `static` définit un membre commun à toutes les instances de la classe
- Le membre est appelé membre de classe
- Une variable de classe est indépendante de l'objet, elle existe dès le premier appel à la classe.
- Une méthode/fonction de classe s'appelle
  - en faisant référence à la classe
  - ou directement à partir d'une autre méthode statique

Exemple de membres de classe (statiques) (2/3)

```
public class Personne {
  String nom;
  int age;
  static int nb = 0;
  int no;

  Personne(){nom=""; age=18; no=nb++;}
  Personne(String nom, int age){
    this.nom = nom; this.age = age; no=nb++;
  }

  /**retourne une chaine contenant le nm, l'age et le no de
      la personne*/
  public String toString(){
    return ("Personne,nom="+nom+",age="+age+ ",no="+no);
  }
}
```

Test de membres de classe (statiques) (3/3)

```
public class TestPersonne {
  static void testStatic()
  {
   var tab = new Personne[3];
   for(int i=0; i<3; i++)
     tab[i] = new Personne("nom"+i, 20+i);
   for(Personne p:tab) System.out.println(p.toString());
  }
  public static void main(String[] args)
  {
   testStatic();
   System.out.println("nb de personnes crees="+Personne.nb);
  }
}
```

→ donne

```
Personne, nom=nom0, age=20, no=0
Personne, nom=nom1, age=21, no=1
Personne, nom=nom2, age=22, no=2
nb de personnes crees=3
```

Classes abstraites (1/3)

Ne pas définir un comportement par défaut

- L'abstraction
  - permet de définir un moule général pour des classes dérivées (*polymorphisme*).
  - Regroupe des noms d'attributs et de méthodes communs à des classes.
- Parfois inutile de donner un corps général à des méthodes,
  - `abstract` devant une méthode permet de ne pas lui donner de corps.
- Si un classe contient au moins une méthode abstraite, elle devient abstraite.
- Une classe abstraite ne peut être instanciée.

Classes abstraites (2/3)

```
abstract class Personne {
  String nom;
  int age;

  Personne(){nom=""; age=18; }
  Personne(String nom, int age){
    this.nom = nom; this.age = age; no=nb++;
  }
  /**declaration de methode sans corps*/
  abstract void affiche();
}
public class Etudiant extends Personne {
  String formation;
  Etudiant(){formation="";}
  Etudiant(String nom, int age, String formation) {
    super(nom, age); this.formation = formation;}
  /**surcharge obligatoire la methode abstraite heritee*/
  @Override
  void affiche() { System.out.println(this.toString());
  }
}
```

Classes abstraites (3/3)

```
//impossibilite de creer une Personne
//possibilite de creer un tableau de Personne
var tab = new Personne[3];

for(int i=0; i<3; i++)
    tab[i] = new Etudiant("nom"+i, 20+i, "UPHF");

//possibilite d'appeler la methode affiche
// car elle est definie comme devant etre instanciee par les
    classes filles
for(Personne p:tab) p.affiche();
```

Interface : une classe "vide" (1/2)

Haut niveau d'abstraction

- Une interface
  - ne contient que les entêtes de méthodes publiques.
  - ne contient que des attributs constants
- Une classe **implémente** une interface.

  ```
  class A implements I1 ...
  ```
- Possibilité d'implémentations multiples.

  ```
  class A implements I1, I2 ...
  ```

  (attention aux conflits pour les attributs identiques)

Interface : une classe "vide" (2/2)

```
public interface EtreVivant {
    void naitre();
}
public interface EtreMortel extends EtreVivant{
    void mourir();
}
public interface EtreSpirituel {
    void philosopher();
}
public class Animal implements EtreMortel {
    public void naitre(){ System.out.println("je naits"); }
    public void mourir() { System.out.println("je meurs"); }
}
public class EtreHumain extends Animal implements
    EtreSpirituel {
    public void philosopher() {System.out.println("je pense
        donc je suis"); }
}
```

Interface : une classe “quasi vide”

Comportement par défaut

Depuis Java 8, une interface peut posséder des corps par défaut pour ses méthodes.

```
interface Individu {
        public default void affiche() {
                System.out.println("je suis un individu");
        }
}
class Personne implements Individu {
    String nom;
    ...
}
```

Interface : implémentations multiples d'interfaces "**quasi** vides"

Implémentations

Le compilateur refuse une implémentation d'interfaces ayant un corps par défaut pour **une même entête**

```
interface Avion {
        public default void bouger() {
                System.out.println("je vole"); }
}
interface Voiture {
        public default void bouger() {
                System.out.println("je roule"); }
}
class VoitureColante implements Avion, Voiture {
// ---> DECLENCHE UNE ERREUR
}
```

Énumération (1/3)

Liste de constantes

- Une énumération est une liste de termes constants :
  `enum Jours {Lundi, Mardi, Mercredi, Jeudi, Vendredi, Samedi, Dimanche}`
- Déclaration d'une variable d'un type énuméré :
  `Jours j = Jours.Samedi`
- Conversion Chaîne $\leftrightarrow$ TypeEnuméré :
  - `j.toString()` retourne la chaîne contenant le terme de la variable *("Samedi")*
  - `Jours j = Jours.valueOf("Samedi")` transforme la chaîne en valeur énumérée, si possible

Énumération (2/3)

Accès aux éléments

- Récupérer l'indice d'une valeur énumérée :
  `int no = j.ordinal();`
- Récupérer les valeurs d'une énumération :

```
Jours[] tab = Jours.values();
System.out.println("Le jour suivant " + j + " est "
+tab[no+1]);
```

→ le jour suivant Samedi est Dimanche

```
//Utilisation dans une selection :
switch(j){
  case Lundi:
  case Mardi:
  case Mercredi:
  case Jeudi:
  case Vendredi: System.out.println("c'est la semaine");
      break;
  case Samedi:
  case Dimanche: System.out.println("c'est le we"); break;
}
```

Énumération (3/3)

Ajouts d'attributs et fonctions

- Une énumération peut contenir attributs (constants) et fonctions :

```
public enum Colis {
  TYPE1(20), TYPE2(30), TYPE3(40);
  /**poids dun colis*/
  int poids;
  /**constructeur obligatoire pour initialiser le poids
      */
  Colis(int _poids){poids = _poids;}
  /**getter pour acceder au poids*/
  public int getPoids(){return poids;}
}
```

- utilisation :

```
Colis c = Colis.TYPE2;
System.out.println("Colis de type " + c + " de poids
    max " + c.getPoids());
```

→ `colis de type TYPE2 de poids max 30`

Record : une classe immubable

Création rapide

Depuis Java 14, un record est une classe dont les éléments sont **constants**.
Sa définition est simplifiée.
Utilisation principalement lors d'extraction de données de liste. Ou pour une aide locale.

```
record MinMax (int min, int max){};

MinMax getExtremes(int ...tab){
        int mini = tab[0];
        int maxi = tab[0];
        for (int v : tab) {
                if (v<mini) mini=v;
                if (v>maxi) maxi=v; }
        return new MinMax(mini, maxi); }
...
MinMax extremes = getExtremes(2,1,4,5,3,6);
System.out.println("min = " + extremes.min);
System.out.println("min = " + extremes.max);
```

Classe interne

Classe imbriquée

- Il est possible d'imbriquer :
  - des classes et interfaces dans des classes
  - des interfaces dans des interfaces
- Une classe interne peut accéder à tous les membres même privés la classe mère

Classe locale

Classe imbriquée dans une méthode

- Une classe locale est définie dans une méthode
- Une classe locale accède aux membres de la classe contenante
- Une classe locale ne peut accéder qu'aux membres de type final de la méthode englobante
- Une classe locale dans une méthode statique ne peut accéder qu'aux membres statiques de la classe contenante

Classe anonyme

Instance de classe sans nom

- Il est possible de créer des classes anonymes héritant d'une classe ou implémentant une interface
- Pour un usage limité à 1 objet pour un comportement original

```
// interface simple
interface Professeur  {
  public String getNom();}


...
// creation d'un objet d'une classe creee dynamiquement
Professeur profAdam = new Professeur(){
  public String getNom(){return "Adam";}
};
System.out.println(profAdam.getNom());
```

→ Adam

Les paquetages

Regroupement de classes

- Un paquetage définit un espace nommé pour un groupe de classes
- Les paquetages sont organisés en hiérarchie
- Importation d'une classe ou de classes d'un paquetage :

```
import NomClass;
import nomPaquetage.NomClasse;
import nomPaquetage1.nomPaquetage2.NomClasse;
import nomPaquetage.*;
```

- Le caractère * ne porte que sur les classes d'un paquetage, pas sur ses sous-paquetages
- Une classe dans un répertoire doit comporter en première ligne le nom du dossier/package :

```
package donnees;
class CompteBancaire{...
```