



Java 11, fondamentaux

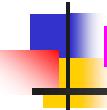
Denis MADEC
dma@magic.fr

v1.5



Plan

- L'approche objet
- Introduction à UML2
- Présentation de Java
- Les types primitifs
- Les structures de contrôle
- Les classes et les objets
- La relation d'association
- La relation de composition
- La relation d'héritage
- Le polymorphisme
- La gestion des exceptions
- Le déploiement d'applications
- Mise en oeuvre de javadoc
- Présentation de la bibliothèque standard
- Les collections



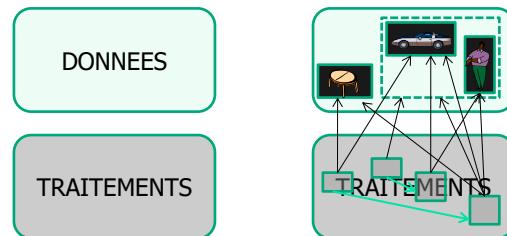
l'approche objet



motivations

- la programmation objet est issue d'un constat de non qualité des logiciels, entraînant un coût exorbitant pour leur maintenance
- la programmation classique s'attache à découper le problème à résoudre en entités fonctionnelles
- les entités fonctionnelles se préoccupent peu des données sur lesquelles elles agissent
 - dans le meilleur des cas, les données sont protégées au sein d'une entité fonctionnelle, car connues d'elle seule
 - dans d'autres cas, les données sont partagées par plusieurs entités fonctionnelles

motivations



Comme en témoigne ce schéma, les données ne sont pas protégées explicitement. Potentiellement, tout traitement peut accéder aux données de l'application ou même à d'autres traitements rendant la lecture et la maintenance de l'application plus compliquées. De même l'évolution du logiciel ne sera pas forcément simple (effet « fondue au fromage »).

Les environnements de développement (IDE) ne peuvent pas proposer sans ambiguïté les traitements possibles pour telle ou telle donnée. Il s'ensuit des efforts de « rangement » et de nommage des données et des traitements pour savoir « qui traite quoi ? » (afficher_personne, afficher_table, afficher_voiture...)

motivations

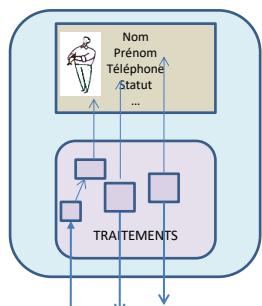
- les concepteurs des langages objet se sont attachés à trouver des solutions pour résoudre les problèmes rencontrés en programmation classique:
 - faible protection des données d'où un grand nombre de bogues potentiels
 - faible réutilisabilité des développements effectués
 - difficulté à modifier des logiciels existants sans compromettre leur fiabilité
 - maintenance rendue difficile par la dépendance des entités fonctionnelles entre elles

principes de l'approche objet

- on cherche à réunifier les deux aspects, données et traitements, pour améliorer:
 - la lisibilité
 - la maintenance
 - l'évolutivité
 - la réutilisabilité

objets

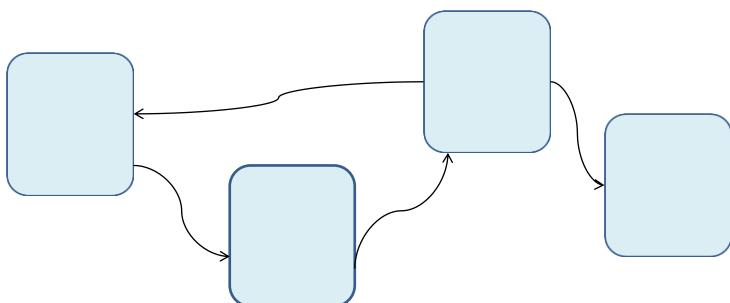
- un objet est une structure qui intègre:
 - des données, appelées champs ou attributs, qui décrivent son état
 - des traitements appelés opérations ou méthodes, qui décrivent son comportement



- les attributs peuvent être des objets
- seules les méthodes sont accessibles depuis l'extérieur

objets

- les objets communiquent entre eux par des messages, complétés ou non par des paramètres
- l'objet récepteur du message déclenche une méthode lui appartenant correspondant au message reçu



objets

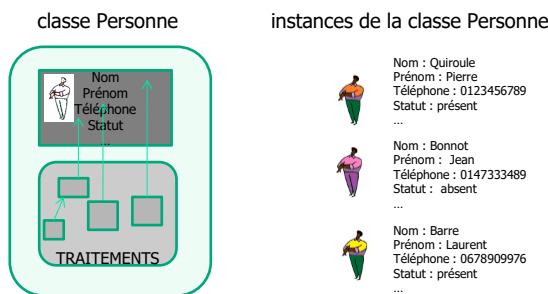
- un objet possède:
 - un état (valeur de ses attributs)
 - un comportement (dicté par ses méthodes)
 - une identité (son nom)

classes

- la structure et le comportement d'objets similaires sont définis dans leur classe commune
 - une classe regroupe donc les objets qui ont un comportement et des données communes
- une classe sert de modèle pour la création d'objets
 - une classe est comme un moule dont seront issus différents exemplaires (ses objets ou instances)
- les objets d'une même classe sont des instances de cette classe
 - les instances peuvent être dans un état différent les unes des autres (couleur, vitesse, nom, ...) mais elles auront le même comportement (mêmes traitements)

classes

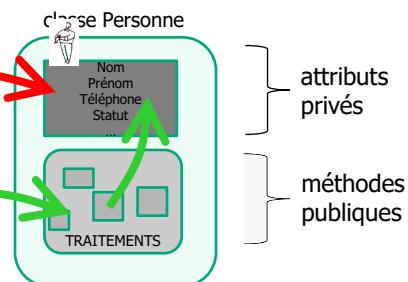
- une classe définit:
 - les attributs
 - les méthodes
 - l'interface de communication avec l'extérieur



classes

- l'encapsulation est le fait d'empaqueter ensemble données et fonctions avec une visibilité réduite depuis l'extérieur
 - l'objectif est de protéger les attributs en interdisant tout accès direct à ces derniers depuis un autre objet situé à l'extérieur

- la mise en oeuvre de l'encapsulation est effectuée dans les classes



classes

- les attributs devraient toujours rester privés
 - la plupart des langages objets autorisent les attributs à être publics, cependant il est recommandé de laisser les attributs privés pour conserver une bonne encapsulation
- les méthodes d'une classe devraient être publiques
 - un message provenant d'un autre objet peut les déclencher
 - elles peuvent dans certains cas rester privées, et sont alors utilisées par des méthodes publiques



classes abstraites

- la modélisation du monde réel doit conduire à un modèle simplifié de ce monde, mais néanmoins le plus proche possible
- or, certains objets de notre monde ne sont pas des objets mais des concepts, comme:
 - animal
 - meuble
 - habitation
- ces concepts sont des factorisations de caractéristiques et/ou comportement d'objets similaires



classes abstraites

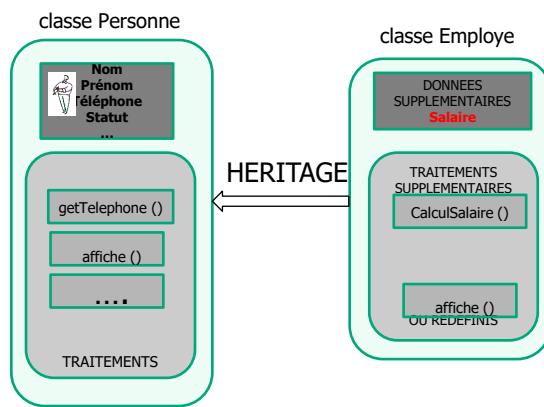
- les classes abstraites permettent de représenter ces concepts:
 - elles ne sont pas instanciables
 - elles servent de super-classes pour des classes concrètes

héritage

- l'héritage traduit la relation EST-UN de notre langage
 - un employé est une personne
 - un employé hérite de personne
 - la relation EST-UN ne doit pas être confondue avec la relation A-UN qui est traduite par la composition
 - une personne a une tête, des jambes, des bras
 - il n'y a pas dans ce cas de relation d'héritage entre tête, jambes ou bras

héritage

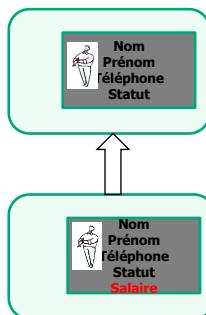
- la classe Personne est appelée super classe, ou classe de base
- la classe Employe est appelée sous-classe, ou classe dérivée



héritage

- l'héritage est mis en oeuvre au niveau des classes, permettant à leurs instances d'en bénéficier

instance de la classe Personne



instance de la classe Employe

- ## héritage
- plusieurs classes peuvent hériter d'une même classe
 - la classe Employe hérite de la classe Personne
 - la classe Actionnaire hérite de la classe Personne
 - une même classe peut hériter de plusieurs classes
 - la classe Directeur hérite de la classe Employe et de la classe Actionnaire car un Directeur EST-UN Employe ET un Actionnaire
 - une sous-classe hérite, c'est-à-dire bénéficie des biens de sa super-classe:
 - de ses attributs
 - de ses méthodes



héritage

- l'héritage est transitif, par conséquent une sous-classe hérite (indirectement) de la super-classe de sa super-classe
- une hiérarchie de classe décrit les relations d'héritage entre plusieurs classes

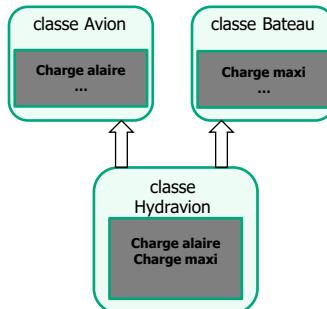


héritage

- l'héritage apporte une technique de réutilisation:
 - ce qui a été développé pour une super-classe peut être réutilisé pour la conception de ses sous-classes
- l'héritage traduit la spécialisation si l'on considère les sous-classes
 - un Employe est une Personne avec un salaire
- il traduit la généralisation si l'on considère les super-classes
 - une Personne représente des caractéristiques et le comportement commun des Employés et des Actionnaires

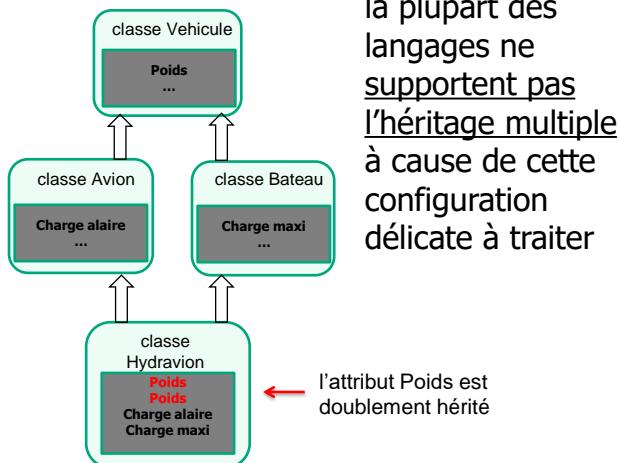
héritage multiple

- l'héritage multiple intervient lorsqu'une classe hérite directement de plus d'une classe



héritage multiple

- l'héritage multiple répété apparaît fréquemment dans le cas de l'héritage multiple



manipulation d'objets

- les instances d'une hiérarchie de classe peuvent être manipulées en les considérant comme du type d'une super-classe

les instances sont toutes considérées comme de type Personne



les méthodes de la classe Personne sont les seules applicables: aucune méthode spécifique aux objets manipulés ne peut être déclenchée



polymorphisme

- le polymorphisme est la propriété qui permet de manipuler des objets sans tenir compte de leur classe
 - cette propriété découle de l'héritage et ne peut être mise en oeuvre qu'à travers ce type de relation
- le polymorphisme permet de manipuler un objet d'une sous-classe en le considérant comme un objet de sa super-classe, tout en conservant sa spécificité (comportement spécialisé)
- la mise en oeuvre du polymorphisme permet de réutiliser du code en apportant des modifications

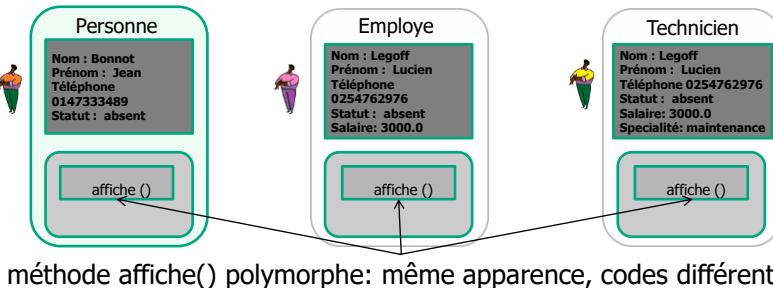
polymorphisme

- l'utilisation de méthodes polymorphes permet de faire apparaître des spécificités liées aux instances manipulées

les instances sont toutes considérées comme de type Personne



l'utilisation de méthodes polymorphes permet alors de manifester leur type propre



méthode affiche() polymorphe: même apparence, codes différents

intérêt de l'approche objet

- outre les avantages d'une meilleure qualité apportée par l'approche objet, celle-ci facilite la modélisation car les objets informatiques sont très proches des objets réels
- l'utilisation d'objets n'exige pas une connaissance de leur structure (comme dans notre monde: conduire une voiture n'exige pas de connaître la mécanique, seulement de savoir conduire)
- l'abstraction (simplification) peut être très poussée
- la manipulation d'objet peut être effectuée par polymorphisme



introduction à UML 2



représentation avec UML

- UML (Unified Modeling Language) est:
 - un langage de modélisation objet
 - constituée d'un ensemble de règles de représentation graphique permettant aux professionnels de communiquer entre eux
- UML propose des moyens, principalement graphiques, de modéliser différents aspects d'un système
- en bref, UML permet d'analyser un système "sous toutes ses coutures"



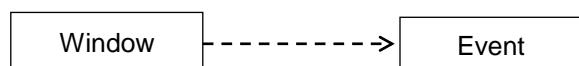
représentation avec UML

- UML propose 13 sortes de diagrammes, dont le diagramme de classes qui permet de représenter les classes et leurs relations
- UML identifie trois types de relations:
 - dépendance
 - généralisation
 - association



diagramme de classes

- la relation de dépendance ou d'utilisation indique le fait qu'une classe en utilise une autre: *Window* utilise *Event*
- cela se traduit par la présence d'un paramètre de type *Event* dans une méthode de *Window*
- *Window* a connaissance d'*Event* mais pas l'inverse



Comparable

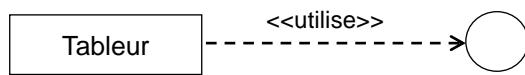


diagramme de classes

- la relation de généralisation ou d'héritage traduit la relation EST UN
- elle indique le fait qu'une classe B est une spécialisation d'une classe A, ou que A est une généralisation de B
- *Employe* a connaissance de *Personne*, mais pas l'inverse

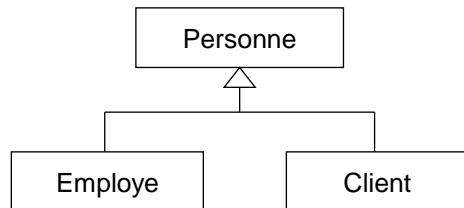


diagramme de classes

- La réalisation traduit le fait qu'une classe implémente une interface (et qu'elle définit donc toutes les méthodes de cette interface)

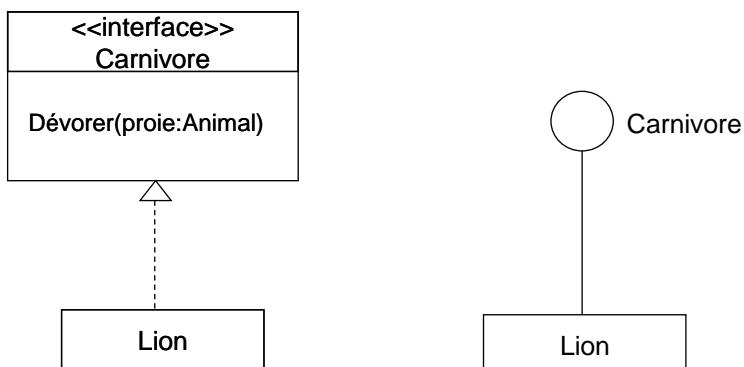




diagramme de classes

- l'association traduit une relation structurelle entre classes:
au moins une des classes impliquée dans l'association a connaissance de l'autre
- l'association peut comporter un nom, ou plutôt une forme verbale, ainsi que la direction vers laquelle elle s'exprime
- elle peut également comporter de part et d'autre le rôle que joue chacune des classes dans la relation
- elle peut de plus comporter de part et d'autre le nombre d'instances impliquées dans l'association



diagramme de classes

- le nombre d'instances impliquées dans une association, de part et d'autre, est de un par défaut
- il est souvent préférable de préciser explicitement la cardinalité (multiplicité):

1 : exactement 1

0..1: zéro ou 1

m..n: de m à n

*: de zéro à plusieurs

1..*: au moins 1

diagramme de classes

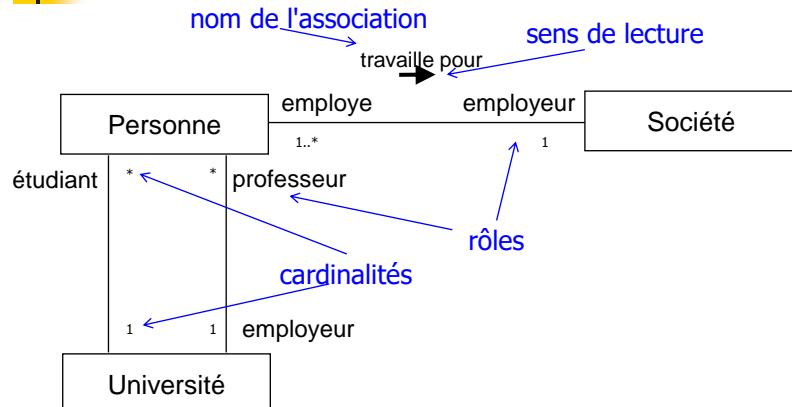
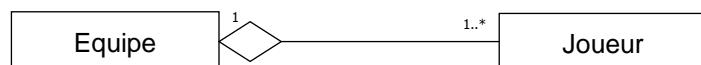


diagramme de classes

- l'agrégation est une forme particulière d'association, dans laquelle une classe joue le rôle d'un tout, l'agrégat, l'autre d'une partie, l'agrégré: elle traduit la relation A UN
 - l'agrégré (ici *Joueur*) peut être lié à d'autres classes et exister sans l'agrégat (ici *Equipe*)



- ◆ une filiale est une société qui peut exister sans sa société mère

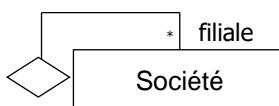




diagramme de classes

- lorsque l'agrégation exprime une contenance physique, elle devient composition:
 - la destruction de l'agrégat entraîne celle de ses agrégés
 - la multiplicité du côté de l'agrégat ne peut prendre que les valeurs 0 ou 1
 - les agrégés ne peuvent être partagés entre agrégats

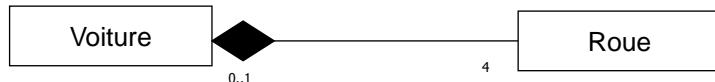
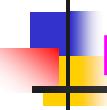


diagramme de classes

- les associations sont par défaut navigables dans les deux sens: chacune des classes a connaissance de l'autre
- navigabilité peut être unidirectionnelle, de sorte qu'une seule des classes a connaissance de l'autre
 - la navigabilité unidirectionnelle est précisée par une flèche



- l'importance de la navigabilité apparaît surtout en phase d'implémentation, car elle permet de simplifier le code



présentation de Java



présentation de Java

- l'approche objet
- **présentation de Java** →
- syntaxe de java
- les classes et les objets
- les tableaux
- le mot-clé this
- les membres static
- arguments et retour de méthodes
- classes, méthodes et variables final
- classes emboîtées
- énumérations
- la relation d'association
- la relation de composition
- la relation d'héritage
- historique
- déclinaisons
- domaines d'utilisation
- le JDK et le JRE
- les outils de développement
- caractéristiques principales
- exemple de programme Java
- compilation et exécution



historique

- Java a été développé chez Sun Microsystems (désormais propriété d'Oracle) en 1991 pour le développement de logiciels enfouis concernant les appareils de grande consommation (télévision, magnétoscopes etc...)
 - il est rapidement devenu un langage de prédilection pour la distribution de programmes exécutables via le World Wide Web (applets) et pris son essor après le développement en 1994 du navigateur HotJava de Sun permettant l'exécution d'applets
 - Java est fortement inspiré de C++ mais certaines fonctionnalités (pointeurs, héritage multiple, surcharge d'opérateurs) ont été exclues.



déclinaisons

- c'est désormais Oracle qui fédère les évolutions du langage
 - selon le type de composant développé, l'environnement d'exécution (runtime) et la bibliothèque associée diffèrent
- trois déclinaisons de Java sont proposées:
 - Java SE: Java Standard Edition
 - il s'agit de la déclinaison de base, celle qui permet le développement d'applications ou d'applets
 - Java EE: Java Enterprise Edition
 - dédiée au développement de composants serveur
 - Java ME: Java Micro Edition
 - il s'agit d'un Java SE allégé, pour l'exécution d'applications sur des machines de puissance réduite (téléphones portables, PDA, etc...)



domaines d'utilisation

- les applications de Java sont aujourd'hui nombreuses, et s'élargissent de jour en jour:
 - applications autonomes avec ou sans interface graphique (Java SE)
 - applications embarquées dans les téléphones portables, les assistants personnels, carte à puce (Java ME)
 - servlets/JSP: substitut aux programmes CGI des serveurs web (Java EE)
 - EJB: composant distribué pouvant être persistant (Java EE)
- Java n'est pour l'instant pas encore utilisé dans les secteurs du temps réel dur, car les contraintes sur la machine virtuelle restent un véritable challenge



le JDK et le JRE

- la société Oracle propose en libre service sur son site www.oracle.com un environnement de développement pour Java appelé JDK: Java Development Kit
 - ce kit comporte de quoi compiler et exécuter des applications Java, ainsi que des outils liés à la sécurité ou aux applications distribuées
 - il suffit d'un simple éditeur pour compléter le JDK et permettre le développement d'applications



le JDK et le JRE

- la version LTS actuelle est JDK 11 (septembre 2018)
 - JDK 10 sorti en 2018
 - JDK 9 sorti en 2017
 - JDK 8 sorti en 2014
 - JDK 7 sorti en 2011
 - JDK 6 sorti en 2006
 - JDK 5 sorti en 2004
 - JDK 1.4 sorti en 2002
- les plateformes disponibles sont:
 - Windows (x64 et x86)
 - Linux (x64 et x86)
 - Solaris (Sparc, x64 et x86)



le JDK et le JRE

- le JRE (Java Runtime Environment) permet seulement d'exécuter des applications Java, sans possibilité de les compiler
 - la version LTS actuelle est JRE 11
- plusieurs versions du JDK et/ou du JRE peuvent cohabiter sur la même machine: il faut dans ce cas porter une grande attention à la version utilisée, sous peine d'incompatibilités



les outils de développement

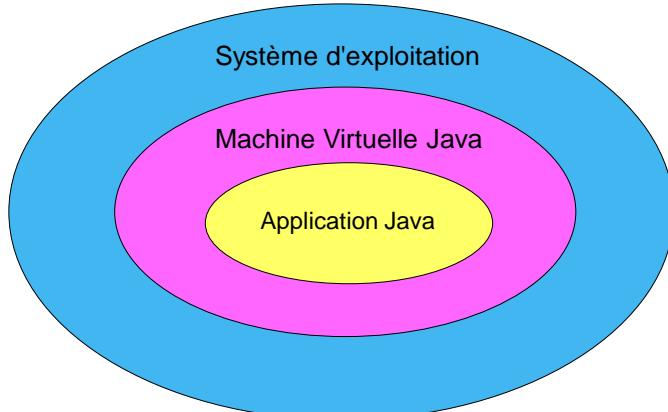
- de nombreux outils de développement Java existent aujourd'hui, notamment en Open Source:
 - Eclipse (Open Source)
 - NetBeans (Open Source)
 - RAD (IBM)
 - IntelliJ IDEA (JetBrains)
- NetBeans et Eclipse permettent le développement Java pour Java SE, et Java EE, Java ME, Android, etc...
- les outils commerciaux sont souvent proposés en plusieurs déclinaisons, offrant des fonctionnalités avancées dans des gammes de prix élevées



caractéristiques principales

- langage indépendant de la plateforme matérielle/logicielle
 - Java est portable au niveau du source et du binaire sur toute plateforme matérielle/logicielle: les fichiers binaires obtenus après compilation peuvent être exécutés sur une machine quelconque sans nécessité d'une recompilation
- Java comporte une bibliothèque de classes standard
 - la bibliothèque de classes est la même quelle que soit la provenance du JRE ou de l'outil de développement
- langage sécurisé
- langage objet

indépendant du système d'exploitation



exemple de programme Java

- soit un fichier source **Bonjour.java** contenant le texte suivant:

```
public class Bonjour {  
    public static void main ( String[] args ) {  
        System.out.println ( "Bonjour à tous" );  
    }  
}
```

- l'exécution de ce programme affiche sur l'écran le message :

Bonjour à tous



compilation et exécution

- la compilation d'un fichier source java (.java) produit un fichier binaire (.class) de byte codes, identique quelle que soit la plateforme matérielle/logicielle:

javac Bonjour.java

- un fichier Bonjour.class est généré par le compilateur javac.exe

- l'exécution du fichier binaire (.class) a lieu dans une machine virtuelle (JVM) qui interprète les byte codes, via un interpréteur spécifique à la plateforme matérielle/logicielle:

java Bonjour

- il s'agit implicitement du fichier Bonjour.class
- le programme java.exe lance la machine virtuelle en précisant la classe qui contient main



compilation et exécution

- lorsqu'une application Java est constitué de plusieurs fichiers sources, il suffit de préciser, en argument de la commande **java**, le nom du fichier .class (sans cette extension) qui contient la méthode **main**
- l'édition de liens est dynamique et transparente pour l'utilisateur (ce sujet est approfondi dans le chapitre sur les packages)



■ Java01: premier programme Java



- l'approche objet
- présentation de Java
- **syntaxe de java** →
 - mots-clés de Java
 - les commentaires
 - les types primitifs
 - les opérateurs et expressions
 - les structures de contrôle
- les classes et les objets
- les tableaux
- le mot-clé this
- les membres static
- arguments et retour de méthodes
- classes, méthodes et variables final
- classes emboîtées
- énumérations
- la relation de composition
- la relation d'héritage
- le polymorphisme



mots-clés du langage

```
char int short byte boolean long float
double
void try catch finally throw private protected
public
transient synchronized native final threadsafe
abstract import
class extends instanceof implements interface
true false this super null new static if else
for do while
switch case default break continue
```



les commentaires

- Java offre trois notations:

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

- identique à celui du C. Le commentaire peut s'étaler sur plusieurs lignes, sans imbrication

```
// commentaire.....
```

- identique à celui du C++. Le commentaire se termine automatiquement en fin de ligne

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

- commentaire **javadoc**, afin de constituer des fichiers de documentation au format HTML



les types primitifs



définition

- ils sont appelés primitifs car ils servent de base à la création d'autres types plus complexes et sont pré-existants dans le langage
 - les variables de ces types ne sont pas des objets
 - leurs caractéristiques sont indépendantes de la machine (portabilité)
- la création d'une variable de type primitif nécessite de préciser son nom et son type, comme:

```
int alpha;           // variable alpha de type int
double beta;         //variable beta de type double
```



les mots-clés

- les types sont précisés par des mots-clés, parmi les suivants:
 - les entiers:

<code>byte</code>	sur 8 bits
<code>short</code>	sur 16 bits
<code>int</code>	sur 32 bits
<code>long</code>	sur 64 bits

- les entiers sont signés et acceptent par conséquent des valeurs positives ou négatives (par exemple le type `byte` accepte les valeurs de -128 à +127)



les mots-clés

- les flottants:

<code>float</code>	sur 32 bits
<code>double</code>	sur 64 bits

- les caractères:

<code>char</code>	sur 16 bits
-------------------	-------------

- dans Java, les caractères sont représentés dans la codification UNICODE, qui représente les caractères sur 16 bits en non signé

- les booléens:

<code>boolean</code>	<code>true</code> OU <code>false</code>
----------------------	---

- le type `boolean` n'est pas un entier et ne peut être utilisé comme tel



les noms de variables

- les noms de variables doivent respecter les règles suivantes:
 - être composés de lettres, du caractère _ (underscore), de caractère monétaires (\$, €..), ou d' un chiffre
 - ne pas commencer par un chiffre
 - Java différencie les majuscules des minuscules



les constantes numériques

- constantes nombres:
 - une constante numérique est représentée par défaut dans le type **int**

```
int i=5;
```
 - ajouter l ou L après la constante force un type **long**

```
long val=5L;
long x=2600356008005L;
```
 - les valeurs négatives sont précédées du signe moins

```
int z=-128;
```



les constantes numériques

- une constante octale commence par 0 (zéro)
`int a=022; // a vaut 18`
- une constante hexadécimale commence par 0x (ou 0X)
`int b=0x22; // a vaut 34`
- une constante décimale comporte un point séparant la partie entière de la partie fractionnaire. La notation scientifique peut également être utilisée avec e ou E
- une constante décimale est représentée par défaut dans le type **double**
`double s=453.67;`
`double u=10e6;`
- ajouter f ou F après la constante force un type **float**
`float t=100.34F;`



les constantes numériques

- depuis Java SE 7, les types entiers (`byte`, `short`, `int`, et `long`) peuvent être exprimés sous forme binaire

- il suffit d'ajouter `0b` ou `0B` devant le nombre

- **exemples:**

```
byte octet = (byte)0b00100001;
short entierCourt = (short)0b1010000101000101;
int entier1 = 0b10100001010001011010000101000101;
int entier2 = 0b101;
int entier3 = 0B101;
long entierLong =
0b1010000101000101101000010110111000101L;
```



les constantes numériques

- le caractère underscore (_) peut apparaître entre les chiffres d'une constante numérique
 - l'intérêt principal est d'améliorer la lisibilité
- les underscores ne peuvent être:
 - être placés au début ou à la fin d'un nombre
 - être adjacent à un point
 - précéder un suffix F ou L
- **exemples:**

```
long carteVisa = 1234_5678_9012_3456L;  
long insee = 2_75_05_78_010_012_07L;  
float pi = 3.14_15F;  
long maxLong = 0x7fff_ffff_ffff_ffffL;  
long bytes = 0b11010010_01101001_10010100_10010010;
```



les constantes numériques

- **constantes booléennes:**
 - elles s'expriment par les mots clés **true** ou **false**
- **constantes caractères:**
 - elles se composent d'une lettre entourée de quotes simples
- **constantes chaînes:**
 - une chaîne est une suite de caractères
 - dans Java, une chaîne est un objet de la classe **String**
 - elles s'expriment par une suite de caractères entourée de quotes doubles

```
String ch="Bonjour";
```



les constantes littérales

- les constantes littérales sont exprimées par le mot-clé **final**, et nécessitent d'être initialisées

```
final double TVA=19.6;
```

```
final int MAX=100;
```

- la valeur d'une constante ne peut être ultérieurement modifiée

```
TVA=20.6; // erreur de compilation
```



affichage et saisie des données

- les méthodes **print** ou **println** de la bibliothèque standard permettent d'afficher des données:

```
int i=74;  
double z=0.375;  
System.out.println(i);  
System.out.println(z);
```

- la méthode **println** fait passer le curseur en début de ligne suivante, alors que **print** laisse le curseur en fin de ligne courante

- lorsque du texte doit être affiché autour d'une valeur numérique, il faut utiliser le signe + (concaténation)

```
System.out.println("valeur de i: "+ i +  
" valeur de z: "+ z);
```



affichage et saisie des données

- les saisies au clavier sont facilitées par la classe **Scanner** de la bibliothèque standard

```
Scanner clavier=new Scanner(System.in);
System.out.print("quantité: ");
int q=clavier.nextInt();
System.out.print("prix: ");
float p=clavier.nextFloat();
```

- les méthodes **nextBoolean**, **nextByte**, **nextShort**, **nextInt**, **nextLong**, **nextFloat**, **nextDouble**, **nextLine**, **next** permettent d'obtenir une donnée typée correspondant au type saisi



atelier

- Java02: affichage de variables



les opérateurs et expressions



opérateur de conversion explicite

- cet opérateur () permet de convertir explicitement un type de donnée dans un autre
 - il est applicable aussi bien aux variables qu'aux constantes (numériques ou littérales)
- appelé également *cast*, il est parfois nécessaire dans certains calculs (voir opérateurs arithmétiques) ou certaines affectations
 - le risque d'une perte d'information doit être pris en compte par le développeur



opérateur de conversion explicite

- **syntaxe:**

```
(type)variable  
(type)constante
```

- **exemple:**

```
double z = 65.87;  
int x = (int)z;           // x vaut 65
```



opérateurs arithmétiques

- Java comporte 5 opérateurs arithmétiques:

- + addition
- soustraction
- *
- / division
- % modulo

- **exemple:**

```
int a=5, b=7, c;  
// . . .  
c = a + b; // c vaut 12 après l'opération
```



opérateurs arithmétiques

- la division doit être traitée avec soin

```
double z = 5/2;  
//division d'entiers->résultat entier  
System.out.println(z); // 2.0  
  
double t = 5.0/2.0;  
//division de flottants->résultat flottant  
System.out.println(t); // 2.5
```



opérateurs arithmétiques

```
int a = 5, b = 2;  
double u = a/b;  
//division d'entiers->résultat entier  
System.out.println(u); // 2.0  
  
double v = (double)a / (double)b;  
//division de flottants->résultat flottant  
System.out.println(v); // 2.5
```



opérateurs sur bits

- Java comporte 7 opérateurs sur bits:

&	ET bit à bit
	OU bit à bit
^	OU exclusif bit à bit
~	complément à 1 (négation)
<<	décalage a gauche
>>	décal. droite arithmétique (conservation du signe)
>>>	décal. droite logique

- exemple:

```
int a=0xCC, mask=0xFFFFFFF8, c;  
// . . .  
c = a & m; // c vaut 0xC8 après l'opération
```



affectations simple et calculée

- affectation simple:

```
x = y;  
z = a = 6;
```

- affectation calculée:

l'opérateur d'affectation est combiné avec un autre opérateur

$x = x \text{ op } y;$ peut s'écrire $x \text{ op=} y;$

- exemple:

```
x += 5;           // idem x = x + 5;  
x &= 0xFFFFF067; // idem x = x & 0xFFFFF067;
```

- il ne faut pas d'espace entre les signes qui composent l'opérateur



affectation calculée

- opérateurs d'affectation calculée:

<code>+=</code>	addition
<code>-=</code>	soustraction
<code>* =</code>	multiplication
<code>/ =</code>	division
<code>%=</code>	modulo
<code>&=</code>	ET bit à bit
<code> =</code>	OU bit à bit
<code>^=</code>	OU exclusif bit à bit
<code><<=</code>	décalage à gauche
<code>>>=</code>	décalage à droite arithmétique
<code>>>>=</code>	décalage à droite logique



opérateurs ++ et --

- ces opérateurs permettent d'incrémenter (++) ou de décrémenter (--) une variable
- la position de l'opérateur a une influence sur le résultat de l'expression.
- on parle de:
 - pré-incrémantation ou pré-décrémantation lorsque l'opérateur est à gauche de la variable
 - post-incrémantation ou post-décrémantation lorsque l'opérateur est à droite de la variable
- exemple:

```
x = ++y; // pré-incrémantation: équivaut à y+=1; x=y;
x = y++; // post-incrémantation: équivaut à x=y; y+=1;
```



opérateurs relationnels

- Java comporte 6 opérateurs relationnels:

==	comparaison égal à
!=	comparaison différent de
<	comparaison strictement inférieur à
>	comparaison strictement supérieur à
<=	comparaison inférieur ou égal à
>=	comparaison supérieur ou égal à

- ces opérateurs fournissent une valeur booléenne

- exemple:

```
int x=7, y=8;  
boolean b = x <= y;      // b vaut true
```



opérateurs logiques

- Java comporte 3 opérateurs logiques:

&&	ET logique
	OU logique
!	négation

- ces opérateurs permettent d'effectuer des opérations de type ET, OU, NON sur des expressions booléennes

- exemple:

```
int x=7, y=8;  
boolean b = x > 0 && x > y;      // b vaut false
```



autres opérateurs

- opérateur ternaire: : ?
- opérateur tableau: []
- opérateur pour la création d'objets: new
- opérateur de test de type d'objets: instanceof



priorité des opérateurs

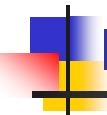
évalués en premier

	associativité
.	de gauche à droite
++ -- ! ~ + - (cast) new	de droite à gauche
*	de gauche à droite
/ %	de gauche à droite
+	de gauche à droite
-	de gauche à droite
<< >> >>>	de gauche à droite
< > >= <= instanceof	de gauche à droite
== !=	de gauche à droite
&	de gauche à droite
^	de gauche à droite
	de gauche à droite
&&	de gauche à droite
	de gauche à droite
?:	de gauche à droite
= += -= *= /= %= ^= &= = ~= <<= >>= >>>=	de droite à gauche



atelier

- Java03: calcul du périmètre d'un cercle



les structures de contrôle



le bloc d'instructions

- un bloc d'instructions est une séquence d'instructions entre accolades
 - les variables définies dans un bloc, ne sont utilisables qu'à l'intérieur de celui-ci, et disparaissent en fin de bloc
 - il n'est pas possible de définir, dans un bloc d'instructions, une variable ayant le même nom qu'une variable définie dans le bloc englobant
- exemple:

```
{  
    int i=5;  
    System.out.println("valeur : " + i);  
    double z=6.9;  
} // les variables i et z n'existent plus!
```



la structure if...else

- la structure de branchement conditionnel **if** ... **else** utilise la valeur d'une expression booléenne pour exécuter ou non des instructions

```
if (expression booléenne) {  
    instructions;  
}
```

- si l'expression est évaluée à vrai, les instructions entre accolades sont exécutées. Elles ne le sont pas sinon

- exemple:

```
if (alpha<10) {  
    alpha++;  
    System.out.println("alpha: " +alpha);  
}
```



la structure if...else

- une autre forme de cette structure est l'alternative:

```
if (expression booléenne) {  
    instructions1;  
} else {  
    instructions2;  
}
```

- si l'expression booléenne est vraie, les instructions1 sont exécutées, sinon les instructions2 sont exécutées

- exemple:

```
if (alpha<10){  
    alpha++;  
    System.out.println("alpha: " +alpha);  
}else{System.out.println("valeur incorrecte ");}
```



la structure if...else

- une troisième variante de cette structure est la suivante:

```
if (expression booléenne1) {  
    instructions1;  
} else if (expression booléenne2) {  
    instructions2;  
} else if (expression booléenne3) {  
    instructions3;  
} else {  
    instructions4;  
}
```

- si l'expression booléenne1 est vraie, alors les instructions1 sont exécutées, sinon, si l'expression booléenne2 est vraie, les instructions2 sont exécutées, sinon, etc...



la structure if...else

- exemple:

```
int choix;  
.  
.  
if (choix==1) {  
    addition();  
} else if (choix==2) {  
    soustraction();  
} else if (choix==3) {  
    multiplication();  
} else if (choix==4) {  
    division();  
} else {  
    System.out.println("choix errone");  
}
```



atelier

- Java04: affichage du signe d'une valeur entière



la structure switch

- la structure précédente peut souvent être remplacée par la structure **switch**:

```
switch(expression entière) {  
    case valeur1: instructions1;  
        break;  
    case valeur2: instructions2;  
        break;  
    case valeur3: instructions3;  
        break;  
    default:       instructions4;  
}
```

- si l'expression entière est évaluée à valeur1, alors les instructions1 sont exécutées, si l'expression entière est évaluée à valeur2, alors les instructions2 sont exécutées



la structure switch

- il peut y avoir autant de cas que nécessaire
- les instructions4 du cas **default** sont exécutées si l'expression entière est évaluée à une valeur non prévue dans les *case*
- l'expression entière doit correspondre à un type de base convertible en **int**
- les valeurs indiquées dans les *case* doivent être des constantes convertibles en **int** (**byte**, **char**, **short** ou **int**)
- l'utilisation d'objets ou de types de base comme **float**, **double** ou **long** est interdite



la structure switch

- exemple:

```
int choix;
. . .
switch(choix) {
    case 1: addition();
              break;
    case 2: soustraction();
              break;
    case 3: multiplication();
              break;
    case 4: division();
              break;
    default: System.out.println("choix errone");
}
```



switch avec String

- la structure **switch** accepte désormais les chaînes de caractères (java 7)

```
public String getHoraire(String jour) {
    String horaire;
    switch (jour) {
        case "Lundi": horaire = "13h - 17h"; break;
        case "Mardi":
        case "Mercredi":
        case "Jeudi":
        case "Vendredi": horaire = "9h - 17h "; break;
        case "Samedi": horaire = "9h - 18h "; break;
        case "Dimanche": horaire = "fermé"; break;
        default:
            throw new IllegalArgumentException("jour invalide");
    }
    return horaire;
}
```



l'opérateur ternaire

- l'opérateur ternaire ou conditionnel ?: nécessite trois opérandes
- il permet d'obtenir l'équivalent d'une structure **if...else** sous forme compacte
- **syntaxe:**
`expression booléenne ? expression2 : expression3`
 - si l'expression booléenne est vraie, alors l'expression2 est évaluée, sinon c'est l'expression3
 - l'opérateur ternaire retourne la valeur de l'expression évaluée
- **exemple:**
`int max = x > y ? x : y;`



la boucle for

- la structure **for** permet de réaliser des boucles puissantes et compactes
- **syntaxe:**
`for (expression1; expression booléenne2 ; expression3) {
 instructions;
}`
 - l'expression1 est évaluée en premier, une seule fois
 - puis l'expression booléenne2 est évaluée, si cette expression est fausse, la boucle prend fin, sinon les instructions entre accolades sont exécutées
 - puis l'expression3 est évaluée
 - l'expression booléenne2 est à nouveau évaluée, etc...



la boucle for

- **exemple:**

```
for (i=0,j=tab.length-1 ; i<j ; i++, j--) {
    System.out.println(tab[i]);
}
```

- l'expression1 correspond en général à l'initialisation du ou des compteurs de boucle
- l'expression3 correspond en général à l'incrémentation du ou des compteurs de boucle
- les accolades sont facultatives si les instructions se réduisent à une seule

```
for (i=0,j=tab.length-1 ; i<j ; i++, j--)
    System.out.println(tab[i]);
```



la boucle for

- les 3 expressions qui figurent entre parenthèses sont facultatives, l'expression booléenne2 est considérée alors comme vraie

- **exemple:**

- boucle infinie:

```
for(;;);
```

- une variable peut être créée à l'intérieur de la boucle et est alors locale à celle-ci

- **exemple:**

```
for (int i=0; i<MAX ; i++) {
    // . . .
} // la variable i n'existe plus
```



la boucle while

- la structure **while** est une autre forme permettant de remplacer la structure **for** dans tous les cas
- **syntaxe:**

```
while (expression booléenne) {  
    instructions;  
}
```

- les instructions sont exécutées tant que l'expression booléenne est vraie
- les initialisations, si elles sont nécessaires, doivent avoir lieu avant le **while**



la boucle while

- **exemple:**

```
i=0;  
while (i< tab.length) {  
    tab[i] = i*i;  
    i++;  
}
```

- les instructions peuvent ne jamais être exécutées si l'expression booléenne est fausse dès l'entrée dans la structure while
- comme pour la boucle **for**, les accolades ne sont pas nécessaires si les instructions se réduisent à une seule



la boucle do...while

- la boucle **do ... while** est très similaire à la boucle **while** mais comporte la particularité d'être exécutée au moins une fois
 - le test est effectué en effet après les instructions du corps de boucle
- **syntaxe:**

```
do {  
    instructions;  
} while (expression booléenne);
```
- ce type de boucle est plus compact que la structure while lorsqu'une itération au moins est nécessaire



la boucle do...while

- **exemple:**

```
DataInputStream dis;  
dis = new DataInputStream(System.in);  
do {  
    . . .  
    System.out.println("Voulez-vous continuer  
(o/n)?");  
    String valStr = dis.readLine();  
} while (valStr.charAt(0) == 'o');
```



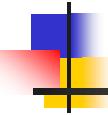
les instructions break et continue

- l'instruction **break** permet une sortie prématuée sans condition d'une boucle

```
for (i=0,j=tab.length-1 ; i<j ; i++, j--) {  
    if (tab[i] != tab[j])  
        break;  
}
```

- l'instruction **continue** permet d'interrompre l'itération en cours et de passer à l'itération suivante

```
for (i=0 ,j=tab.length-1 ; i<j ; ) {  
    if ((c1=tab[i++])== SPACE) continue;  
    if ((c2=tab[j--])== SPACE) continue;  
    if (c1 != c2) break;  
}
```



ateliers

- Java05: mise en œuvre d'une boucle while
- Java06: mise en œuvre d'une boucle do...while
- Java07: mise en œuvre d'une boucle for
- Java08 (optionnel): calcul d'une factorielle - type de boucle au choix



les classes et les objets

- l'approche objet
- présentation de Java
- syntaxe de java
- **les classes et les objets** →
 - introduction
 - la compilation de classes
 - la définition des classes
 - l'instanciation des classes
 - l'appel des méthodes
 - l'initialisation des attributs
 - les propriétés
 - les packages et les classes
 - l'accès aux membres
 - la surcharge de méthodes
 - les constructeurs et la méthode finalize
 - les chaînes de caractères
 - les opérations sur les objets
 - les conversions de types
- les tableaux
- le mot-clé this
- les membres static
- arguments et retour de méthodes
- classes, méthodes et variables final
- classes emboîtées
- énumérations
- la relation d'association
- la relation de composition
- la relation d'héritage



introduction

- une classe est un modèle pour la création d'objets
- elle comporte des données membre ou attributs ou champs d'instance
- elle comporte des fonctions membre ou méthodes
- elle définit l'interface d'accès avec l'extérieur pour ses membres
- les attributs d'une classe peuvent être des types de base, des objets
- par convention, le nom d'une classe commence par une majuscule



la compilation des classes

- par convention, on admet une classe et une seule par fichier source
 - ce fichier source a pour nom celui de la classe qui y est définie, suivi de l'extension **.java**
- le compilateur génère alors un fichier de byte codes ayant pour nom celui de la classe, suivi de l'extension **.class**
 - en réalité, seule les classes publiques doivent obligatoirement se trouver dans un fichier du même nom
 - il peut y avoir plusieurs classes non publiques dans un même fichier source. Le compilateur génère alors autant de fichiers **.class** que de classes présentes dans le fichier source



la définition de classe

Compteur.java

```
public class Compteur {          // nom de la classe
    private int cpt;              // attribut

    public void increment() {     // méthode
        ++ cpt;
    }
    public void affiche() {       // méthode
        System.out.println ("valeur : " + cpt);
    }
    public int lirecount() {      // méthode
        return cpt;
    }
}
```



l'instanciation d'une classe

- "instancier" une classe signifie créer un objet sur le modèle de cette classe
 - un objet est une instance de classe
- tous les objets de cette même classe auront un comportement identique et les mêmes types d'attributs
 - chaque objet comportera ses propres attributs
 - les objets d'une même classe pourront avoir des états différents



l'instanciation d'une classe

- il faut tout d'abord créer une variable du type de la classe:

```
Compteur c1;
```
- il faut ensuite créer un objet de la classe avec l'opérateur **new** et l'affecter à la variable précédemment définie:

```
c1 = new Compteur ();
```
- plus simplement, il suffit d'écrire:

```
Compteur c1 = new Compteur ();
```



l'appel des méthodes

- une méthode doit être appelée (invoquée) sur un objet donné:

Appli.java

```
public class Appli{  
    public static void main ( String args[] ) {  
        Compteur c1 = new Compteur ();  
        c1.affiche();  
        c1.increment());  
        int x = c1.lirecount();  
        System.out.println ("Valeur : " + x);  
    }  
}
```

- lorsqu'un objet n'est plus référencé, la mémoire qu'il occupe dans le tas est récupérable. le Garbage Collector (ramasse-miettes) est un élément de la JVM qui gère cette récupération mémoire, de façon automatique



initialisation des attributs

- **new** initialise automatiquement les attributs d'un objet à:

- 0 pour les valeurs numériques
- false pour les booléens
- '\0' pour les caractères
- null pour les autres

- il est néanmoins possible de fournir explicitement des valeurs aux attributs:

```
public class Compteur {  
    private int cpt=1; // attribut initialisé  
    . . .
```



les propriétés

- de façon à faciliter l'accès en lecture/écriture aux attributs, un couple de méthodes publiques est très souvent associé à chaque attribut
- par convention, ces méthodes sont nommées `getXxxx` et `setXxxx` pour permettre respectivement l'accès en lecture et en écriture à un attribut nommé `xxxx`



les propriétés

- exemple:

```
public class Personne{  
    private String nom;  
    . . .  
    public String getNom(){  
        return nom;  
    }  
    public void setNom(String n) {  
        nom=n;  
    }  
}
```



les propriétés

- *Xxxx* devient alors une propriété par le couple de méthodes *getXxx/setXxx*
- *nom* est une propriété d'une instance de la classe *Personne*
- une seule des méthodes peut être fournie, limitant ainsi l'accès à une propriété
- le nom de la propriété est donné par le nom des méthodes, pas par celui de l'attribut associé
- de nombreux standard et frameworks s'appuient sur cette convention pour fonctionner: JSP, JSF, JPA/Hibernate, etc....



atelier

- Java09: création et instanciation de la classe Compteur



les packages et les classes



objectif

- pour éviter les conflits, chaque classe devrait avoir un nom différent
 - cela peut être difficile à éviter lorsque le nombre de classes est important
- dans Java, un package permet de gérer les espaces de noms
 - il est ainsi possible de définir des classes de même nom à condition qu'elles appartiennent à des packages différents
- par défaut, une classe ne fait partie d'aucun package, ce que l'on nomme package par défaut
- les classes d'un même package ont des relations privilégiées entre-elles



mot-clé package

- l'appartenance d'une classe à un package est définie par le mot-clé **package** suivi du nom du package, en tout début de fichier source (avant les instructions **import**)

```
package util;  
public class Appli{ // la classe Appli fait partie  
                    // du package util  
  
    }  
■ le nom d'une classe inclut celui de son package:  
    util.Appli
```



mot-clé package

- le compilateur Java exige qu'une classe soit placée dans le répertoire de même nom que le package dont elle fait partie
- le fichier **Appli.java** doit nécessairement se trouver dans un répertoire nommé **util**
- l'arborescence de répertoires suit donc celle des packages



mot-clé package

- la structure de la bibliothèque de classes Java est organisée en packages, sous forme arborescente
 - par défaut, les classes accèdent seulement aux classes du package `java.lang` de la bibliothèque, à celles du package par défaut, ainsi qu'à celles du package dont elles font partie, s'il y a lieu
- une classe d'un package ne peut être utilisée dans un autre package que si la classe est déclarée **public**:

```
package entreprise;  
public class Personne {  
  
    .  
    .  
}
```



utiliser les classes

- pour utiliser les classes d'un autre package, il faut les référencer explicitement avec le nom complet du package ou les importer dans le fichier source
 - référencer une classe :

```
java.awt.Color c = new java.awt.Color(...);
```

- importer une classe:

```
import java.awt.Color;
```

- il est parfois plus commode de généraliser l'import à toutes les classes du package:

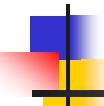
```
import java.awt.*;
```

remarque: seules les classes peuvent être importées, et par conséquent, le symbole * ne peut se rencontrer plus d'une fois dans une clause **import**



packages et classpath

- l'utilisation de package(s) lors du développement d'un projet suppose que le compilateur et la machine virtuelle sachent retrouver les classes qui s'y trouvent
 - ce problème est en grande partie géré par les environnement de développement, mais le déploiement des classes suppose de savoir le résoudre de toutes façons
- les outils `javac`, `java`, `javadoc` utilisent le « *user class path* » pour trouver les classes, autres que celles de la bibliothèque java, utilisées par une application
 - le « *user class path* » indique par défaut le répertoire courant



packages et classpath

- une variable d'environnement `CLASSPATH` permet de modifier le comportement par défaut, en indiquant la liste des répertoires à explorer pour trouver les répertoires correspondants aux packages

```
C:\> set CLASSPATH=path1;path2...
```
- l'option `-classpath` de `javac` ou de `java` joue le même rôle que cette variable mais se trouve d'un emploi plus aisé

```
C:\> java -classpath path1;path2...
```

 - l'option `-classpath` prédomine si la variable `CLASSPATH` est aussi utilisée



packages et classpath

- soit une classe `Demo`, sans indication de package, se trouvant dans le répertoire `exo`, celui-ci n'étant pas le répertoire courant:
- le « *user class path* » doit indiquer le chemin du répertoire `exo`:

```
set CLASSPATH=C:\devlp\exo; ...
java Demo
```

ou

```
java -classpath C:\devlp\exo Demo
```



packages et classpath

- soit une classe `Test`, avec indication du package `usine.essai`
- si le chemin menant au répertoire `usine` est `C:\devlp`, et si le répertoire courant n'est pas `C:\devlp`, alors le « *user class path* » devrait l'indiquer:

```
set CLASSPATH=C:\devlp; ...
java usine.essai.Test
```

ou

```
java -classpath C:\devlp usine.essai.Test
```



packages et classpath

- soit une classe Personne, avec indication du package entreprise, se trouvant dans un fichier C:\java\devlp\gestion.jar
- le « *user class path* » doit indiquer le chemin du fichier gestion.jar:

```
set CLASSPATH=C:\java\devlp\gestion.jar; ...
java entreprise.Personne
```

ou

```
java -classpath C:\java\devlp\gestion.jar
entreprise.Personne
```



atelier

- Java10: définition des classes Compteur et Appli au sein d'un package propre



l'accès aux membres

- les méthodes d'une classe ont toujours directement accès aux autres membres de cette même classe (variables ou méthodes)
- l'interface de la classe définit l'accès à ses membres depuis l'extérieur de la classe, c'est-à-dire depuis une autre classe
- l'interface d'accès aux membres d'une classe est défini par les mots-clés:
`public, protected, private`
- ces mots-clés sont à placer devant les membres de la classe



l'accès aux membres

- **public**
 - un membre `public` est accessible de tout autre objet d'une autre classe
- **protected**
 - un membre `protected` est accessible de tout autre objet d'une classe du même package, inaccessible de tout objet d'une classe d'un autre package sauf d'une classe dérivée
- **private**
 - un membre `private` interdit son accès depuis tout objet d'une autre classe



l'accès aux membres

- ***accès par défaut***

- un membre sans mot-clé `public`, `protected` ou `private` est accessible de tout autre objet d'une classe du même package, inaccessible d'un objet d'une classe d'un autre package
 - ce type d'accès est également appelé *package-private*



l'accès aux membres

accès	depuis une classe quelconque du même package	depuis une classe quelconque d'un autre package	depuis une sous-classe
<code>private</code>	non	non	non
<code>protected</code>	oui	non	oui
<code>public</code>	oui	oui	oui
<i>par défaut</i>	oui	non	non



l'accès aux membres

- un bon programme orienté objets doit conserver l'encapsulation des données
 - les variables doivent donc rester privées dans la mesure du possible
 - les méthodes permettant d'y accéder seront publiques
- bien que non recommandé, il est possible d'accéder aux attributs, si l'interface défini dans la classe le permet:

```
c1.cpt = 0; // ok si cpt est accessible
```



la surcharge de méthodes

- la surcharge de méthodes consiste à donner le même nom à des méthodes différentes dans une même classe

```
public class Date {  
    private int jour;  
    private String mois;  
    private int annee;  
  
    public void setDate (int j, String m, int a) {...}  
  
    public void setDate (int numero_jour) {...}  
}
```

- il s'agit d'une facilité d'écriture



la surcharge de méthodes

- le compilateur différencie les méthodes lors de l'appel par le nombre et le type des arguments

```
public class TestDate {  
    public static void main(String[] args){  
        int jour;  
        Date d1=new Date();  
        . . .  
        d1.setDate(30, "juin",2000); setDate(int, String, int)  
        . . .  
        d1.setDate(128);  
    }  
}
```

- le type de retour n'intervient pas dans la surcharge
- il n'y a pas de limites aux nombres de surcharges d'une même méthode



atelier

- Java11: ajout de méthodes surchargées dans la classe Compteur



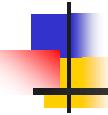
les constructeurs

- 
- ## les constructeurs
- un constructeur est une méthode automatiquement invoquée lors de la création d'un objet
 - un constructeur n'est invoqué qu'une seule fois dans la vie d'un objet
 - il sert essentiellement à initialiser les attributs
 - bien que non indispensable, un constructeur permet la création d'un objet en précisant des valeurs initiales pour ses attributs
 - un constructeur doit respecter les règles suivantes:
 - son nom est obligatoirement celui de sa classe
 - il ne doit pas comporter de type de retour (même void)
 - il ne peut être appelé explicitement



les constructeurs

- un constructeur se doit d'être accessible depuis la méthode dans laquelle on crée l'objet . Il est en général *public*
- les constructeurs peuvent être surchargés
- en l'absence de tout constructeur explicite, le langage fournit un constructeur implicite appelé "constructeur par défaut"
 - néanmoins, s'il existe un constructeur avec argument(s), la construction d'un objet sans argument nécessitera alors la présence d'un constructeur sans argument



les constructeurs

- exemple:

```
public class Date {  
    private int jour;  
    private String mois;  
    private int annee;  
  
    public Date() {  
        //. . .  
    }  
    public Date (int j, String m, int a) {  
        //. . .  
    }  
    public Date (int numero_jour) {  
        //. . .  
    }  
}
```



les constructeurs

- exemple (suite):

```
public class TestDate {  
    public static void main(String args[]) {  
        Date d1=new Date();  
        Date d2=new Date(30,"octobre",2018);  
        Date d3=new Date(302);  
    }  
}
```



les constructeurs

- un constructeur ne peut être explicitement invoqué pour un objet, il s'agit d'une méthode réservée pour le compilateur
- un constructeur peut invoquer un constructeur de la même classe par la syntaxe:
`this (arg1, arg2, ...);`
 - le compilateur recherche dans la classe un constructeur qui comporte le nombre et le type des arguments précisés entre parenthèses
 - seul un constructeur peut appeler un autre constructeur
 - cette instruction, si elle est présente dans un constructeur, doit en être la première



atelier

- Java12: ajout de constructeurs dans la classe Compteur



la méthode finalize



la méthode finalize

- si elle est présente dans sa classe d'un objet, la méthode **finalize** est automatiquement invoquée par la machine virtuelle juste avant la disparition de cet objet

```
public void finalize() {...}
```

- elle peut parfois être utilisée lorsque certaines ressources telles que des fichiers, des sockets qu'il faut libérer sont utilisés dans l'objet
- l'appel à cette méthode par la machine virtuelle n'est jamais garanti



les chaînes de caractères

- l'utilisation d'une constante chaîne entraîne la création automatique d'un objet de la classe **java.lang.String** avec la valeur fournie:

```
String s1="Bonjour ";
// idem: String s1=new String ("Bonjour ");
```

- chaîne vide:

```
String sv="";
// idem: String sv=new String();
```



les chaînes de caractères

- pour simplifier l'utilisation des chaînes de caractères, les opérateurs + et += permettent de les concaténer

```
System.out.println (s1 + "a tous!");  
// Bonjour a tous !  
s1 += "Monsieur";  
// idem: s1 = s1 + "Monsieur"
```

- les opérateurs + et += permettent également de concaténer une chaîne avec un nombre :

```
int x=10;  
String s= "valeur: " + x;
```



les chaînes de caractères

- la classe **java.lang.String** de la bibliothèque standard comporte des méthodes pour manipuler les chaînes:

```
public class String {  
    public String();  
    public String(String ch);  
    public String(StringBuffer chb);  
    public String(char[] s);  
    public char charAt(int index);  
    public int compareTo(String ch);  
    public boolean equals(Object obj);  
    public int length();  
    public char[] toCharArray();  
    public String toLowerCase();  
    public String toUpperCase();  
    public String trim();  
    public String substring(int deb, int fin);  
    public int indexOf(int c);  
    public int indexOf(String ch);  
}
```



les chaînes de caractères

- les instances de **String** sont toujours des chaînes de caractères constantes
- par exemple, une méthode telle que **toUpperCase** ne modifie pas l'objet **String** concerné mais en crée un autre comportant les caractères en majuscules. L'objet initial, s'il n'est plus référencé, sera traité par le ramasse-miettes
- pour des raisons d'efficacité, les programmes qui effectuent de nombreuses opérations sur les chaînes de caractères ont alors intérêt à plutôt manipuler des objets de la classe **StringBuilder**



les chaînes de caractères

- il est parfois nécessaire de modifier une chaîne de caractères
- la classe **StringBuilder** comporte des méthodes permettant notamment d'ajouter, d'insérer ou de supprimer des caractères:

```
public class StringBuilder {  
    public StringBuilder ();  
    public StringBuilder (String ch);  
    public StringBuilder (longueur);  
    public char charAt(int index);  
    public int length();  
    public StringBuffer append(String ch);  
    public StringBuffer append(char c);  
    public StringBuffer insert((int offset, String ch);  
    public StringBuffer insert(int offset, char c);  
    public void setCharAt(int index, char c);  
    public void setLength(int longueur);  
    public String toString();  
}
```



ateliers

- Java13: détection de palindrome
- Java14: formatage en majuscule-minuscules d'un nom propre



les opérations sur les objets

- seuls les opérateurs suivant ont une signification pour des objets:
 - = (affectation)
 - == (comparaison égal à)
 - != (comparaison différent de)
- ces opérateurs affectent ou comparent des références aux objets, non les objets eux-mêmes



les opérations sur les objets

```
Compteur c1 = new Compteur(5);
Compteur c2 = new Compteur(5);
if (c1 == c2) { // test faux, c1 et c2 sont deux références
                // sur deux objets distincts

if (c1.compare (c2)) { // test vrai, c1 et c2 sont deux
// références sur deux objets distincts identiques

c1=c2;
if (c1 != c2) { // test faux, c1 et c2 sont deux références
                // sur le même objet
c2 = new Compteur ();
if (c1 != c2) { // test vrai, c1 et c2 sont deux références
                // sur deux objets distincts
```



atelier

- Java15: ajout d'une méthode de comparaison à la classe Compteur



les conversions de types

- conversions entre types primitifs:

- d'un type vers un type plus grand ne pose pas de problème:

```
int x;  
double y = x;
```

- d'un type vers un type plus petit peut entraîner une perte d'information.: le cast explicite est dans ce cas nécessaire:

```
int a = 1000;  
byte z = (byte) a;
```

- les booléens ne peuvent pas être convertis

- conversions entre objets

- les conversions entre des objets ne peuvent avoir lieu que si leurs classes sont liées par héritage



les conversions de types

- conversions entre types primitifs et objets

- il n'existe pas de conversion directe entre un type primitif et une classe
 - l'autoboxing (depuis Java 5) fait apparaître certaines affectations comme des conversions naturelles, grâce aux classes enveloppes
 - le package **java.lang** contient en effet des classes spéciales appelées classes enveloppes (wrapper classes) qui correspondent à chacun des types primitifs:
 - **Boolean, Character, Short, Integer, Long, Float, Double**
 - les méthodes de ces classes permettent entre autres de créer un objet à partir d'un type de base et d'obtenir la valeur de cet objet dans un type de base



autoboxing

- avant Java 5:

```
int x=15;  
Integer objx = new Integer(x);  
. . .  
int v = objx.intValue();
```

- depuis Java 5:

```
int x=15;  
Integer objx = x;           // autoboxing  
. . .  
int v = objx;              // unboxing
```

- cette fonctionnalité doit être employée à bon escient, car elle masque la création d'objets



les tableaux

- l'approche objet
- présentation de Java
- syntaxe de java
- les classes et les objets
- **les tableaux** →
 - tableaux de types primitifs
 - tableaux d'objets
 - tableaux à plusieurs dimensions
- le mot-clé this
- les membres static
- arguments et retour de méthodes
- classes, méthodes et variables final
- classes emboîtées
- énumérations
- la relation d'association
- la relation de composition
- la relation d'héritage



tableaux de types primitifs

- un tableau est un ensemble de données consécutives de mêmes types
- définir un tableau en Java consiste à:
 - définir une référence précisant le nom et le type du tableau:

```
int [] tab; // ou int tab[];
```
 - créer un objet tableau par `new` et l'affecter à la référence précédemment définie:

```
tab = new int[10];
```

 - la taille du tableau peut figurer en variable
 - plus simplement, il suffit d'écrire:

```
int [] tab = new int [10];
```



tableaux de types primitifs

- les indices de tableaux sont contrôlés à l'exécution:

```
tab[0]=3;
tab[10]=5; // tab[10] n'appartient pas au tableau
//exception de type ArrayIndexOutOfBoundsException
```
- pour éviter les exceptions à cause d'un indice erroné lors de la manipulation d'un tableau, il est conseillé d'utiliser la variable `length` qui représente la longueur (nombre d'éléments) d'un tableau donné:

```
for (int i = 0 ; i < tab.length ; i++){
    System.out.println (tab[i]);
}
```



tableaux de types primitifs

- une autre solution pour parcourir les tableaux (et les collections) est d'utiliser une boucle *foreach*

```
for (int x : tab){  
    System.out.println (x);  
}
```

- ce type de parcours ne permet pas de modifier un élément du tableau



tableaux de types primitifs

- **new** initialise automatiquement les éléments d'un tableau à:
 - 0 pour les valeurs numériques
 - false pour les booléens
 - '\0' pour les caractères
- un tableau peut être explicitement initialisé lors de sa création:

```
int [] tab = { 5, 8, 9, 1, 0, 3};
```

- la taille du tableau est alors automatiquement déterminée par le compilateur en fonction du nombre d'éléments indiqués entre accolades



tableaux d'objets

- définir un tableau d'objets consiste à:
 - définir une référence précisant le nom et le type du tableau:

```
Compteur [] tc;           // ou Compteur tc[];
```

- créer un objet tableau et l'affecter à la référence précédemment définie:

```
tc = new Compteur [10];
```

- plus simplement, il suffit d'écrire:

```
Compteur [] tc = new Compteur [10];
```

- un tableau d'objets ne stocke pas directement des objets mais des références aux objets
 - il convient donc de l'initialiser avec des objets



tableaux d'objets

- **new** initialise automatiquement les éléments d'un tableau d'objets à **null**

```
Compteur c1 = new Compteur();  
tc[0] = c1;  
tc[2] = new Compteur(5);  
tc[3] = new Compteur(100);  
. . .  
for (int i = 0 ; i < tc.length ; i++){  
    tc[i].affiche(); // exception NullPointerException  
                           // lors de l'accès à tc[1]  
}
```



tableaux d'objets

- un tableau d'objets peut être explicitement initialisé lors de sa création:

```
Compteur [] tc = {new Compteur(4), new Compteur(2),  
                   new Compteur() };  
  
String[] jours =  
    {"Lundi", "Mardi", "Mercredi", "Jeudi",  
     "Vendredi", "Samedi", "Dimanche"};
```

- la taille du tableau est alors automatiquement déterminée par le compilateur en fonction du nombre d'éléments indiqués entre accolades



tableaux d'objets

- l'affectation d'un tableau ou d'un élément du tableau à un autre n'effectue pas la copie d'un tableau ou d'un objet mais seulement la copie des références

```
String[] jours = {"Lundi", "Mardi", "Mercredi", "Jeudi",  
                  "Vendredi", "Samedi", "Dimanche"};  
  
String[] j = jours;  
// j et jours désignent le même tableau  
// il n'y a pas recopie des éléments de jours dans j  
  
jours[0] = jours[6];  
// jours[0] et jours[6] sont deux références  
// sur le même objet Dimanche  
// Dimanche n'est pas copiée dans jours[0]
```



les tableaux à plusieurs dimensions

- les tableaux à plusieurs dimensions se déclarent comme tableaux de tableaux:

```
int courbe[][] = new int[10][20];
courbe[0][0] = 0;
courbe[1][19] = 150;
```

- comme il s'agit d'un tableau de pointeurs, le nombre de colonnes peut ne pas être précisé lors de la définition:

```
int tab[][] = new int[10][]; // tableau à 2 dimensions de 10 lignes
tab[0] = new int[20]; // la première ligne est un tableau de 20 int
tab[1] = new int[5]; // la deuxième ligne est un tableau de 5 int
tab[2] = 4; // erreur à la compilation, 4 n'est pas de type int[]
```



les tableaux à plusieurs dimensions

- l'initialisation explicite d'un tableau à plusieurs dimensions ressemble à celle d'un tableau à une seule dimension:

```
int courbe[][] = {{4, 5, 8, 0, 6},
                  {5, 6, 1, 9, 128, 89, 0, 7},
                  {-42000, 96, -5}};

for (int i=0 ; i<courbe.length ; i++) {
    for (int j=0 ; j<courbe[i].length ; j++) {
        println("Element [ " + i + " ] [ " + j +
               " ] == " + courbe[i][j]);
    }
}
```



atelier

- Java16: création, initialisation, affichage d'un tableau d'entiers et tableau d'objets Compteur



le mot-clé this

- l'approche objet
- présentation de Java
- syntaxe de java
- les classes et les objets
- les tableaux
- **le mot-clé this** →
 - définition
 - utilisation de this
- les membres static
- arguments et retour de méthodes
- classes, méthodes et variables final
- classes emboîtées
- énumérations
- la relation d'association
- la relation de composition
- la relation d'héritage



définition

- **this** est un mot-clé qui représente l'objet courant, c'est-à-dire l'objet pour lequel la méthode a été invoquée, au sein d'une méthode d'instance
 - **this** est une référence à l'objet courant
- dans une méthode d'instance, il peut être utilisé pour:
 - différencier les attributs de variables locales de mêmes noms
 - retourner l'objet courant
 - le passer en argument à une autre méthode
 - éviter des traitements inutiles ou impossibles



utilisation de this

- utilisation de **this** pour différencier les attributs de variables locales de mêmes noms:

```
public class Compteur {  
    private int cpt=0;           // attribut  
  
    public void init(int cpt) {  
        this.cpt=cpt;           // cpt désigne l'argument  
                               // this.cpt désigne l'attribut  
    }  
    public void affiche() {  
        System.out.println ("valeur : " + cpt);  
    }  
}
```



utilisation de **this**

- utilisation de **this** pour retourner l'objet courant:

```
public class Compteur {  
    private int cpt;  
    public Compteur increment() {  
        cpt++;  
        return this;      // après incrémentation, l'objet  
                           // courant est retourné  
    }  
    public void affiche() { System.out.println(cpt); }  
}  
  
public class Test {  
    public static void main(String [] args) {  
        Compteur c1 = new Compteur();  
        c1.increment().affiche(); // valeur: 1  
    }  
}
```



utilisation de **this**

- utilisation de **this** pour transmettre l'objet courant en argument:

```
public MonApplet extends Applet implements Runnable {  
    private Thread th;  
    .  
    .  
    public void start() {  
        if(th==null) {  
            th=new Thread(this);  
            th.start();  
        }  
    }  
}
```



utilisation de **this**

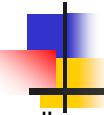
- utilisation de **this** pour éviter des traitements inutiles ou impossibles:

```
public class Personne {  
    . . .  
    public void mariage(Personne p) {  
        if(this != p){  
            // traitement  
        }else{  
            // erreur: on ne peut se marier avec soi-même  
        }  
    }  
}
```



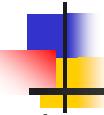
atelier

- Java17: retour de l'objet courant par quelques méthodes de la classe Compteur



les membres static

- l'approche objet
- présentation de Java
- syntaxe de java
- les classes et les objets
- les tableaux
- le mot-clé this
- **les membres static** →
 - les variables de classe
 - les méthodes de classe
 - la méthode main
 - import static
 - les blocs d'initialisation
- arguments et retour de méthodes
- classes, méthodes et variables final
- classes emboîtées
- énumérations
- la relation d'association
- la relation de composition
- la relation d'héritage



les variables de classe

- les variables de classe sont "globales" à une même classe, et n'existent qu'en un seul exemplaire
- leurs valeurs sont les mêmes pour tous les objets de cette classe
- elles ne se retrouvent pas dans chacun des objets de cette classe
- il s'agit donc de variables partagées par tous les objets d'une classe
- comme les variables d'instance, elles bénéficient des accès private, protected, public et celui par défaut



les variables de classe

```
public class Article {  
    private static int nombreArticles = 0; // variable de classe  
    private int code = 0; // attribut  
    private String designation = "XXXXXX"; // attribut  
  
    .  
    .  
}
```



les variables de classe

- lorsqu'une variable de classe est en accès **public**, elle peut être invoquée depuis l'extérieur de la classe de deux façons:
 - via un objet de cette classe comme pour un attribut **public**:

```
Article a1 = new Article();  
a1.monnaie="F";
```
 - via le nom de la classe (solution préférable):
Article.monnaie="F";
- l'accès à une variable de classe depuis une méthode de cette classe s'effectue comme pour un attribut



les méthodes de classe

- les méthodes de classe ne concernent pas un objet en particulier mais plutôt la classe elle-même, ou bien tous les objets de cette classe
- le mot-clé **static** définit une méthode de classe
- une méthode de classe ne peut pas utiliser **this**, puisqu'elle n'est pas liée à un objet donné
- elle ne peut pas accéder directement (sans préciser un objet) aux attributs ni aux méthodes d'instance, pour la même raison
- elle peut accéder aux variables de classe (c'est son rôle principal) ou à d'autres méthodes de classe



les méthodes de classe

```
public class Article {  
    private static int nombreArticles = 0; // variable de classe  
    private int code = 0; // attribut  
    private String designation = "XXXXXX"; // attribut  
  
    public static int lire_nombreArticles () { // méthode  
        // de classe  
        return nombreArticles;  
    }  
    . . .  
}
```



les méthodes de classe

- les méthodes de classe peuvent être invoquées de deux façons différentes:

- via un objet de cette classe comme pour une méthode d'instance publique:

```
Article a1 = new Article();
System.out.println("Nombre d'articles : " +
    a1.lire_nombreArticles());
```

- via le nom de la classe (solution préférable):

```
System.out.println("Nombre d'articles : " +
    Article.lire_nombreArticles());
```

- l'accès à une méthode de classe depuis une méthode d'instance de cette classe peut s'effectuer directement



la méthode main

- la méthode **main** est nécessairement une méthode de classe publique
- la méthode **main** est le point d'entrée dans une application Java
 - la classe correspondant au programme java à exécuter doit comporter une méthode **main**
- un programme java constitué de plusieurs classes pourrait comporter plusieurs méthodes **main**
 - la méthode **main** exécutée est celle indiquée en argument de l'interpréteur **java**



la méthode main

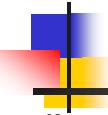
- il est possible de passer des paramètres à l'application lors de son lancement:

```
java Appli fic1 5 "version 3"
```

- **main** reçoit un tableau de chaînes de caractères en arguments

```
public class Appli {  
    public static void main (String args[]) {  
        for (int i = 0 ; i < args.length ; i++ ) {  
            System.out.println ("arg"+i+ " : " + args[i]);  
            // arg 0: fic1  
            // arg 1: 5  
            // arg 2: version3  
        }  
    }  
}
```

- les paramètres sont transmis sous la forme de chaînes de caractères qu'il faut parfois convertir en valeurs numériques pour les exploiter facilement



imports static

- l'accès à un membre déclaré **static** nécessite de le faire précéder de son nom de classe:

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

- de façon à simplifier le code, les membres **static** peuvent être importés au moyen d'un **import static**:

```
import static java.lang.Math.PI;  
.  
.  
.  
perim=2*rayon*PI;
```

- les membres **static** peuvent être importés en masse:

```
import static java.lang.Math.*;
```



les blocs d'initialisation

- une variable de classe peut être initialisée de plusieurs façons:
 - en lui donnant une valeur initiale lors de sa définition

```
private static int nombreArticles = 0;
```
 - en utilisant un bloc d'initialisation « **static** », lorsque l'initialisation nécessite l'exécution de plusieurs instructions

```
public class Alpha {  
    private static int[] tab=new int[32];  
    static {           // bloc d'initialisation  
        int puis=1;  
        for(int i=0 ; i<tab.length ; i++, puis*=2) {  
            tab[i]=puis;  
        }  
    }  
    ...
```



les blocs d'initialisation

- les blocs d'initialisation **static** ne peuvent accéder qu'aux variables ou aux méthodes de classe
- il peut y avoir autant de blocs d'initialisation **static** que nécessaire
- ces blocs d'instructions sont exécutés au premier chargement de la classe, après l'initialisation explicite des variables de classe, dans l'ordre de leur position à l'intérieur de la classe
 - ❖ la méthode **main** est exécutée après l'exécution de tous les blocs déclarés **static**



atelier

- Java18: comptage d'objets Compteur au moyen d'une variable de classe



arguments et retour de méthodes

- l'approche objet
- présentation de Java
- syntaxe de java
- les classes et les objets
- les tableaux
- le mot-clé this
- les membres static
- **arguments et retour de méthodes** → • passage d'arguments aux méthodes
• retour de méthodes
- classes, méthodes et variables final
- classes emboîtées
- énumérations
- la relation d'association
- la relation de composition
- la relation d'héritage



le passage d'arguments

- les arguments des types de base sont transmis aux méthodes par valeur (ou par copie)
 - une méthode appelée ne peut modifier la donnée de la méthode appelante
- les arguments de type tableau ou objet sont transmis aux méthodes par référence:
 - toute modification de ces arguments dans la méthode appelée se répercute sur les objets initiaux de la méthode appelante



le passage d'arguments

```
public class Appli {  
    public static void main (String args[]) {  
        int i = 5;  
        int result=carre(i); // la variable i est  
        // transmise par valeur. Apres l'appel, i vaut 5  
        // result vaut 25  
    }  
    private static int carre(int c) {  
        c *= c;  
        return c;// la variable c est retournée par valeur  
    }  
}
```



le passage d'arguments

```
public class Appli {  
    public static void main (String args[]) {  
        int [] tab = new int [10];  
        init(tab);  
        for (int i=0 ; i<tab.length ; i++)  
            System.out.println(tab[i]);// affiche 0, 1, 2 ,...  
            // le tableau tab a été modifié  
    }  
    private static void init(int[] t) {  
        for (int i=0 ; i<t.length ; i++)  
            t[i]=i;  
    }  
}
```



le passage d'arguments

```
public class Appli {  
    public static void main (String args[]) {  
        Counter c1=new Counter(5);  
        incr(c1); // l'objet c1 est transmis par référence  
        c1.affiche(); // apres l'appel, c1 est modifié  
                    //(son attribut vaut 6)  
    }  
    private static void incr(Counter c) {  
        c.increment();  
    }  
}
```



le retour d'objets

- les données des types primitifs sont retournées par valeur
 - il faut donc réaliser une affectation pour récupérer la valeur renvoyée
- les tableaux ou objets renvoyés par les méthodes le sont par référence



le retour d'objets

```
public class Compteur {  
    private int cpt;  
    public Compteur add(int c)  {  
        cpt+=c;  
        return this;          // retour de l'objet courant  
    }  
    public void affiche() {  
        System.out.println ("valeur : " + cpt);  
    }  
}  
public class Appli {  
    public static void main (String args[]) {  
        Compteur c1=new Compteur (5);  
        c1.add(4).add(6);           // le retour d'objets par référence  
                                // permet l'associativité  
        c1.affiche();              // affiche 15  
    }  
}
```



les classes, méthodes et variables déclarées final

- l'approche objet
- présentation de Java
- syntaxe de java
- les classes et les objets
- les tableaux
- le mot-clé this
- les membres static
- arguments et retour de méthodes
- **classes, méthodes et variables final** → • les classes final
• les méthodes final
• les variables final
- classes emboîtées
- énumérations
- la relation d'association
- la relation de composition
- la relation d'héritage



les classes final

- une classe dont on ne veut pas créer de classe(s) dérivée(s) peut être définie avec le mot-clé **final**:

```
public final class Maclasse {  
    .  
    .  
    .  
}  
public class SousClasse extends MaClasse { // ERREUR !  
    .  
    .  
    .  
    // héritage refusé par le compilateur  
}
```



les méthodes final

- une méthode dont on ne souhaite pas pouvoir créer de méthodes polymorphes peut être définie avec le mot-clé **final**:

```
public class Maclasse {  
  
    public final void affiche() {  
  
        . . .  
    }  
}
```

- les méthodes final ont pour principal intérêt d'être optimisées (équivalentes aux méthodes inline du C++) car traitées sous la forme de macro-instructions



les variables final

- la plupart des constantes sont définies comme membres **static** d'une classe

```
public class Article {  
  
    private static final double TVA=19.6;  
  
    . . .  
}
```

- un attribut constant doit nécessairement être initialisé explicitement ou dans chacun des constructeurs

```
public class Article {  
    private final double TVA;  
    public Article(double tva){  
        TVA = tva;  
    }  
    . . .  
}
```



les variables final

- lorsqu'une référence est déclaré en **final**, cette référence ne peut désigner un autre objet, mais l'objet désigné peut être modifié

```
public class Article {  
    . . .  
  
    public void calculePrix(final Ristourne r){  
        r = new Ristourne(); // erreur de compilation !  
        r.ajout(5);          // OK  
    }  
    . . .  
}
```



classes emboîtées

- présentation de Java
- syntaxe de java
- les classes et les objets
- les tableaux
- le mot-clé this
- les membres static
- arguments et retour de méthodes
- classes, méthodes et variables final
- classes emboîtées** →
 - présentation
 - classe emboîtée static (*static nested class*)
 - classe interne (*inner class*)
 - classe interne locale
 - classe anonyme
- énumérations
- la relation d'association
- la relation de composition
- la relation d'héritage



présentation

- une classe emboîtée (*nested class*) est une classe définie à l'intérieur d'une autre classe:
 - soit comme membre **static** de cette classe
 - soit comme membre de cette classe, au même titre que ses attributs ou méthodes
 - une classe peut même être définie à l'intérieur d'une méthode de cette classe
- les classes emboîtées permettent:
 - de regrouper ensemble des classes étroitement liées
 - d'améliorer l'encapsulation
 - de rendre le code plus compact et plus facile à maintenir



classe emboîtée *static*

- la classe emboîtée peut être déclarée avec le mot-clé **static**
- ```
public class A {
 public static class B {
 }
}
```



## classe emboîtée *static*

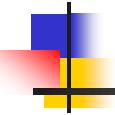
- une classe emboîtée **static** fonctionne comme une classe ordinaire
  - elle n'est pas rattachée à une instance de sa classe englobante
  - elle ne peut accéder qu'aux variables et méthodes **static** de la classe englobante
  - elle peut bénéficier des accès **private**, **protected**, **public** et par défaut
  - depuis l'extérieur à la classe englobante, le nom de la classe emboîtée doit être préfixé par celui de sa classe englobante



## classe emboîtée *static*

- **exemple:**

```
public class Coords {
 private final Pair[] array;
 public Pair add(int index,int x,int y) {
 return array[index]=new Pair(x,y);
 }
 public static class Pair {
 private final int x,y;
 ...
 }
}
public class Appli{
 public static void main(String[] args) {
 Coords coords=new Coords(10);
 Coords.Pair pair=coords.add(0,1,2);
 }
}
```

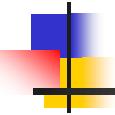


## classe interne

- une classe interne (*inner class*) est définie comme membre de la classe

```
public class A {
 public class B {
 }
}
```

- une classe interne définie comme membre d'une classe subit les mêmes règles de visibilité que les autres membres
  - les spécificateurs d'accès **private**, **protected**, **public** s'appliquent donc
- une classe interne est associée à une instance de sa classe englobante
  - elle a un accès direct aux membres de cette dernière



## classe interne

- exemple:

```
public class Sequence {
 private final char[] array;

 public class Sub {
 private final int offset;
 private final int length;
 public char charAt(int index) {
 if (index<0 || index>=length)
 throw new IllegalArgumentException(...);
 return array[offset+index];
 }
 }
}
```



## classe interne

- exemple (suite):

- la création d'une instance de la classe interne doit être effectuée sur une instance de sa classe englobante

```
public class Appli{
 public static void main(String[] args) {
 Sequence seq = new Sequence("toto");
 Sub sub = seq.new Sub(1,3);
 System.out.println(sub.charAt(0));
 }
}
```



## classe interne

- exemple (suite):

- à l'intérieur de la classe englobante, **this** est pris par défaut

```
public class Sequence {
 private final char[] array;

 public class Sub {
 . . .
 }
 public Sub subsequence(int offset) {
 return new Sub(offset, array.length-offset);
 // return this.new Sub(...);
 }
}
```



## classe interne

- une classe interne a accès à l'instance de sa classe englobante avec la syntaxe:

`OuterClass.this`

- exemple:

```
public class Holder {
 ...
 public void print() {
 System.out.println(this);
 }
 public class Printer {
 public void print() {
 System.out.println(this);
 System.out.println(Holder.this);
 }
 }
}
```



## classe interne locale

- une classe interne locale est définie à l'intérieur d'une méthode

- ne peut être déclarée ni `static` ni `private` ni `protected` ni `public`

```
public class A {
 public void m() {
 class B {
 }
 }
}
```



## classe interne locale

- une classe interne locale peut utiliser les variables et paramètres locaux de la méthode, mais seulement s'ils sont déclarés **final**
  - l'instance de la classe locale contient en fait une copie des variables, et afin d'éviter d'avoir deux variables modifiables avec le même nom et la même portée, elles doivent être constantes



## classe interne locale

- exemple:

```
public class MaFenetre extends JFrame {
 . . .
 public MaFenetre(final String titre) {
 class EcouteurFenetre extends WindowAdapter{
 public void windowClosing(WindowEvent e){
 System.out.println("fermeture de "+titre);
 System.exit(0);
 }
 }
 EcouteurFenetre ef=new EcouteurFenetre();
 addWindowListener(ef);
 }
}
```



## classe interne anonyme

- une classe interne anonyme est définie lorsque son instance est créée

```
public class A {
 public void m() {
 new Object() {...};
 }
}
```

- la classe ainsi créée n'a pas de nom, elle hérite ou implémente de type indiqué après `new`
- la classe interne anonyme ne peut avoir de constructeur explicite
- ces classes sont très utilisées, notamment dans la gestion d'évènements et dans la création de threads



## classe interne anonyme

- exemple:

```
public class MaFenetre extends JFrame {
 . . .
 public MaFenetre() {
 addWindowListener(new WindowAdapter() {
 public void windowClosing(WindowEvent e){
 System.exit(0);
 }
 });
 }
}
```



## énumérations

- présentation de Java
- syntaxe de java
- les classes et les objets
- les tableaux
- le mot-clé this
- les membres static
- arguments et retour de méthodes
- classes, méthodes et variables final
- classes emboîtées
- **énumérations** →
  - intérêt
  - définition
  - utilisation
  - énumérations complexes
- la relation d'association
- la relation de composition
- la relation d'héritage



## intérêt

- dans certains cas, les constantes forment une liste consécutives de valeurs

```
public static final int LUNDI=0;
public static final int MARDI=1;
public static final int MERCREDI=2;
public static final int JEUDI=3;
public static final int VENDREDI=4;
public static final int SAMEDI=5;
public static final int DIMANCHE=6;
```

- inconvenients:

- le compilateur ne détecte pas la présence de deux valeurs identiques
- l'affichage des constantes apparaît comme une valeur entière



## définition

- Java propose un moyen de déclarer des énumérations:

```
enum Semaine {LUNDI,MARDI,MERCREDI,JEUDI,
 VENDREDI,SAMEDI,DIMANCHE};

Semaine jour;

. . .
jour=Semaine.LUNDI;
System.out.println(jour); // LUNDI
```

- déclarer une énumération revient à créer une liste d'instance (LUNDI, MARDI, etc...) d'une classe (Semaine) qui implémente les interfaces **Serializable** et **Comparable**
- les énumérations ne peuvent pas être déclarées en variable locale



## utilisation

- elles peuvent être affichées, traitées dans un `switch`

```
switch(jour){
 case LUNDI: . . .
 break;
 case MARDI: . . .
 break;
 case MERCREDI: . . .
 break;
 case JEUDI: . . .
 break;
 case VENDREDI: . . .
 break;
 case SAMEDI: . . .
 break;
 case DIMANCHE: . . .
 break;
}
```



## utilisation

- une énumération comporte par défaut une méthode `static values()` qui retourne toutes les valeurs de l'énumération sous la forme d'un tableau:

```
// boucle ordinaire
for(int i=0;i<Semaine.values().length;i++) {
 System.out.println(Semaine.values()[i]);
}

// boucle for each:
for(Semaine j: Semaine.values()) {
 System.out.println(j);
}
```



## énumérations complexes

- une énumération peut comporter des champs, des méthodes et des constructeurs

```
public enum SystemState {
 OFF("le système est éteint"),
 ON("le système est en marche"),
 SUSPEND("le système est en arrêt");
 private String description;

 private SystemState(String d) { // constr. private
 description = d;
 }
 public String getDescription() {
 return description;
 }
}
```



## la relation d'association

- présentation de Java
- syntaxe de java
- les classes et les objets
- les tableaux
- le mot-clé this
- les membres static
- arguments et retour de méthodes
- classes, méthodes et variables final
- classes emboîtées
- énumérations
- **la relation d'association** → • définition  
• mise en oeuvre
- la relation de composition
- la relation d'héritage



## définition

- la relation d'association traduit la relation "A-UN" ou "POSSEDE-UN"
  - cela se traduit par la présence d'un objet partie en attribut de l'objet composé
  - la navigabilité peut être unidirectionnelle, de sorte qu'une seule des classes a connaissance de l'autre
    - la navigabilité unidirectionnelle est précisée par une flèche





## mise en oeuvre

- **exemple:**

```
public class Cercle {
 private Point centre;
 private double rayon;
 public Cercle(){
 this.centre = new Point();
 this.rayon=0;
 }
 public Cercle(Point centre, double rayon){
 this.centre = centre;
 this.rayon=rayon;
 }
 public void deplace(double dx, double dy){
 centre.deplace(dx, dy);
 }
 public void setCentre(Point centre){
 this.centre = centre;
 }
 public Point getCentre() { return centre; }
 . . .
}
```



## mise en oeuvre

- **exemple:**

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

 public Point(){
 this(0,0);
 }
 public Point(double x, double y){
 this.x=x;
 this.y=y;
 }
 . . .
}
```



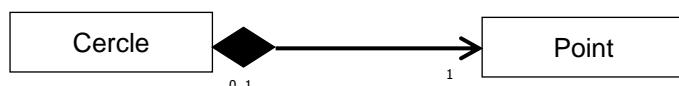
## la relation de composition

- présentation de Java
- syntaxe de java
- les classes et les objets
- les tableaux
- le mot-clé this
- les membres static
- arguments et retour de méthodes
- classes, méthodes et variables final
- classes emboîtées
- énumérations
- la relation d'association
- **la relation de composition** → • définition  
• mise en oeuvre
- la relation d'héritage



## définition

- la relation de composition traduit la relation "A-UN" ou "POSSEDE-UN"
  - cela se traduit par la présence d'un objet partie en attribut de l'objet composé



- l'objet composé doit initialiser la référence sur cet objet attribut
  - car une référence possède la valeur `null` par défaut
  - cette initialisation aura lieu dans le constructeur de l'objet composé



## définition

- la composition lie la durée de vie des objets en relation
  - la création de l'objet composé entraîne en principe celle de l'objet partie
  - la destruction de l'objet composé entraîne en principe celle de l'objet partie
  - l'objet partie ne peut être partagé par d'autres objets composés



## mise en oeuvre

- exemple:

```
public class Cercle {
 private Point centre;
 private double rayon;
 public Cercle(){
 this.centre = new Point();
 this.rayon=0;
 }
 public Cercle(Point centre, double rayon){
 this.centre = new Point(centre);
 this.rayon=rayon;
 }
 public void deplace(double dx, double dy){
 centre.deplace(dx, dy);
 }
 public void setCentre(Point centre){
 this.centre = new Point(centre);
 }
 public Point getCentre() { return new Point(centre); }
}
```



## mise en oeuvre

- **exemple:**

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

 public Point(){
 this(0,0);
 }
 public Point(double x, double y){
 this.x=x;
 this.y=y;
 }
 // constructeur de copie
 public Point(Point p){
 this.x=p.x;
 this.y=p.y;
 }
 . . .
}
```



## atelier

- Java19: développement d'une classe Personne en relation d'association, puis de composition avec une classe Adresse (travail préparatoire à l'héritage)



## la relation d'héritage

- présentation de Java
  - syntaxe de java
  - les classes et les objets
  - les tableaux
  - le mot-clé this
  - les membres static
  - arguments et retour de méthodes
  - classes, méthodes et variables final
  - classes emboîtées
  - énumérations
  - la relation d'association
  - la relation de composition
  - **la relation d'héritage**
- 
- définition
  - implémentation de l'héritage
  - construction d'un objet de sous-classe
  - l'appel des méthodes
  - l'accès aux membres d'une super-classe
  - la manipulation d'un objet de sous-classe



## définition

- l'héritage consiste à faire bénéficier une classe (les objets de cette classe) des attributs et des méthodes d'une autre classe
- l'héritage entre classes s'exprime par le mot-clé **extends**

```
public class B extends A {
 .
 .
}
```

- B est sous-classe (ou classe dérivée) de A
- A est super-classe (ou classe de base) de B
- plusieurs classes filles peuvent hériter d'une même classe parente



## implémentation de l'héritage

- l'héritage traduit la relation "EST-UN"
  - l'héritage multiple n'est pas implémenté dans Java
    - une forme d'héritage multiple est néanmoins réalisée par le biais des interfaces
- l'héritage est transitif et conduit à une hiérarchie d'héritage
  - une sous-classe peut ainsi avoir plusieurs super-classes, au travers des relations d'héritage qui peuvent exister entre ces dernières



## implémentation de l'héritage

- en plus des méthodes publiques de sa classe, il est possible d'invoquer les méthodes publiques de sa(ses) super-classe(s)
  - un objet d'une sous-classe est en général "plus gros" qu'un objet de sa super-classe, car en plus des attributs définis dans sa propre classe, il comporte ceux de sa(ses) super-classe(s)
  - il a de plus un comportement plus varié qu'un objet de sa super-classe, puisqu'il dispose de toutes les méthodes héritées, en plus de celles qu'il possède dans sa propre classe



## construction d'un objet de sous-classe

- l'initialisation des attributs d'un objet de sous-classe peut être effectuée en faisant appel aux constructeurs, lesquels peuvent appeler ceux de la classe de base par la syntaxe:

```
super (arg1, arg2,);
```

- le compilateur ira rechercher dans la classe de base immédiate s'il existe un constructeur qui correspond au nombre et au type des arguments entre parenthèses
- si elle est présente dans un constructeur, l'instruction:  

```
super(arg1, arg2,...);
```

doit en être la première instruction



## construction d'un objet de sous-classe

```
public class Personne {
 private String nom;
 public Personne() {nom = "";}
 public Personne (String n) { nom = n; }
}
public class Employe extends Personne {
 private double salaire;
 public Employe() {
 super(); // facultatif car implicite.
 // appelle le constructeur Personne()
 salaire=0.0;
 }
 public Employe (String n, double s) {
 super (n); // appelle le constructeur Personne(String)
 salaire = s;
 }
}
```



## atelier

- Java20: définition d'une classe Employe, sous-classe de Personne



## l'appel des méthodes

- lorsqu'une méthode est invoquée pour un objet, le compilateur recherche d'abord cette méthode dans la classe de cet objet
- si aucune méthode de ce nom n'est présente dans la classe, il remonte dans la hiérarchie de classes pour continuer sa recherche
- toutes les méthodes publiques des super-classes peuvent ainsi être invoquées pour un objet d'une sous-classe, en plus des méthodes publiques propres à cette classe



## l'appel des méthodes

```
public class Personne {
 private String nom;
 public Personne (String n) { nom = n; }
 public void affiche_pers () {
 System.out.println ("nom : " + nom); }
 }
 public class Employe extends Personne {
 private double salaire;
 public Employe (String n, double s) {
 super (n); salaire = s; }
 }
 public class Appli {
 public static void main (String args[]) {
 Employe martin=new Employe ("MARTIN", 15000);
 martin.affiche_pers (); // méthode héritée
 // de la classe Personne
 }
 }
```



## l'appel des méthodes

- si l'on souhaite compléter le code d'une méthode héritée, il est avantageux de définir une méthode qui appelle la méthode héritée:

```
public class Personne {
 private String nom;
 public Personne (String n) { nom = n; }
 public void affiche_pers () {
 System.out.println ("nom : " + nom); }
 }
 public class Employe extends Personne {
 private double salaire;
 public Employe (String n, double s) { super (n); salaire = s; }
 public void affiche_emp () {
 affiche_pers (); // appel d'une méthode héritée
 System.out.println ("salaire : " + salaire); }
 }
 }
```



## l'appel des méthodes

- dans le cas où les méthodes appelantes et appelées portent le même nom, il faut indiquer au compilateur qu'il ne s'agit pas d'une méthode récursive en faisant précéder l'appel à la méthode du mot-clé **super**

```
public class Personne {
 private String nom;
 public Personne (String n) { nom = n; }
 public void affiche () { System.out.println ("nom : " + nom);}
}
public class Employe extends Personne {
 private double salaire;
 public Employe (String n, double s) { super (n); salaire = s; }
 public void affiche () {
 super.affiche(); // appel d'une méthode héritée de même nom
 System.out.println ("salaire : " + salaire);
 }
}
```



## atelier

- Java21: définition d'une classe Technicien, sous-classe de Employe



## l'accès aux membres d'une super-classe

- si l'on souhaite encapsuler les données, il est souhaitable de les rendre privées dans la classe
  - une méthode d'une sous-classe n'a pas directement accès aux attributs définis dans sa super-classe: elle doit passer par des méthodes publiques de sa super-classe
- l'encapsulation peut être conservée, tout en autorisant l'accès aux données depuis les méthodes des sous-classes, en déclarant certains membres **protected** dans la super-classe
  - ces membres restent privés pour un objet de la super-classe
    - les membres **protected** sont vus comme s'ils étaient déclarés **public** pour les classes du package (voir tableau d'accès)



## l'accès aux membres d'une super-classe

```
package a;
public class Employe extends Personne {
 protected double salaire;
 public Employe (String n, double s) { super (n); salaire = s; }
 public void setSalaire (int s) {
 salaire = s;
 }
}
package c;
public class Technicien extends Employe {
 private String specialite;
 protected int echelon;
 .
 .
 public void fixer_salaire (int val) {
 salaire = val; // accès à un attribut
 //protected dans Employe
 }
}
```



## l'accès aux membres d'une super-classe

```
package b;
public class Usine {
 public static void main (String args[]) {
 Empoye Lucien = new Empoye ("Lucien", 17000);
 Lucien.salaire = 20000; // ERREUR ! salaire est protected
 Lucien.setSalaire (20000); // OK
 }
}
```



## atelier

- Java22: mise en place d'un accès protected



## la manipulation d'un objet de sous-classe

- un objet d'une sous-classe peut être implicitement converti en objet d'une super-classe dans la hiérarchie:

```
Employe martin = new Employe ("MARTIN", 15000);
Personne p = martin; // martin est une Personne
```

- il est par exemple possible de passer en paramètre à une méthode un objet qui sera reçu dans un argument du type d'une de ses super-classes

```
public class Appli {
 public static void main (String args[]) {
 Employe martin=new Employe ("MARTIN", 15000);
 majus(martin);
 }
 private static void majus(Personne p) {
 .
 }
}
```



## la manipulation d'un objet de sous-classe

- l'héritage autorise donc la manipulation d'objets de sous-classes en les considérant comme des objets de leur super-classe

```
public class Compta {
 public static void main (String args[]) {
 Personne durand = new Personne ("DURAND");
 Employe martin = new Employe ("MARTIN", 15000);
 durand.affiche_pers (); // nom : DURAND
 martin.affiche_emp (); // nom : MARTIN
 // salaire : 15000
 Personne p = martin; // l'objet martin
 // est une Personne
 p.affiche_pers (); // nom : MARTIN
 }
}
```



## la manipulation d'un objet de sous-classe

- un objet d'une sous-classe peut nécessiter d'être manipulé via une référence de sa propre classe, de façon à pouvoir utiliser les méthodes (non polymorphes) de sa classe
- un transtypage (cast) est alors nécessaire pour y parvenir
  - une exception **ClassCastException** est lancée par la JVM si le type de l'instance ne correspond pas au type du cast



## la manipulation d'un objet de sous-classe

- exemple:

```
public class Compta {
 public static void main (String args[]) {
 Personne durand = new Personne ("DURAND");
 Personne martin = new Employe ("MARTIN", 15000);
 Employe e2=(Employe)martin; // ok à la compilation
 // ok à l'exécution
 e2.fixer_salaire(3000);

 Employe e1=(Employe)durand; // ok à la compilation
 // ClassCastException
 .
 .
 .
 }
}
```



## la manipulation d'un objet de sous-classe

- afin d'éviter un **ClassCastException**, il est conseillé de faire appel à l'opérateur **instanceof**
- exemple:

```
public class Compta {
 public static void main (String args[]) {
 Personne durand = new Personne ("DURAND");
 Personne martin = new Employe ("MARTIN", 15000);
 if(martin instanceof Employe){ // true
 Employe e2=(Employe)martin;
 e2.fixer_salaire(3000);
 }
 if(durand instanceof Employe){ // false
 Employe e1=(Employe)durand;
 e2.fixer_salaire(3000);
 }
 }
}
```



## atelier

- Java23: affichage détaillé des objets Personne, Employe, Technicien contenus dans un tableau



## le polymorphisme

- **le polymorphisme** →
  - les classes et méthodes abstraites
  - les interfaces
  - les exceptions
  - compléments
  - outils jar et javadoc
  - la bibliothèque standard
  - les collections
- objectif
  - méthodes redéfinies
  - annotation @Override



## objectif

- l'héritage autorise la manipulation d'objets de sous-classes en les considérant comme des objets de leur super-classe (relation "EST-UN"), cependant ils perdent alors leur spécificité d'objet de sous-classe
- le polymorphisme permet de manipuler des objets sans se soucier de leur classe (liées néanmoins par héritage)
- si l'on souhaite manipuler des objets de sous-classe en les considérant comme des objets de leur super-classe, tout en bénéficiant des spécificités qui leurs sont rattachées, il faut définir des méthodes polymorphes



## méthodes redéfinies

- des méthodes polymorphes ou redéfinies sont des méthodes appartenant à des classes liées par héritage et qui ont le même prototype (nom, type de retour, arguments)



## méthodes redéfinies

- exemple:

```
public class Personne {
 private String nom;
 public void affiche () {
 System.out.println ("nom : " + nom);
 }
}

public class Employe extends Personne {
 private double salaire;
 public void affiche() { //méthode polymorphe ou redéfinie
 super.affiche(); // appel de la méthode héritée
 System.out.println ("salaire : " + salaire);
 }
}
```



## méthodes redéfinies

```
public class Compta {
 static public void main (String args[]) {
 Personne Durand = new Personne ("DURAND");
 Employe Martin = new Employe ("MARTIN", 15000);
 Durand.affiche(); // nom : DURAND
 Martin.affiche(); // nom : MARTIN
 // salaire : 15000
 Personne p = Martin; // l'objet Martin est
 // une Personne
 p.affiche(); // nom : MARTIN
 // salaire : 15000
 }
}
```



## méthodes redéfinies

- les méthodes polymorphes peuvent ne pas être redéfinies dans les sous-classes.
  - dans ce cas, ce sera la méthode polymorphe de la super-classe immédiatement au-dessus qui sera utilisée
- pour mettre en oeuvre le polymorphisme, la méthode polymorphe doit être soit héritée, soit déclarée dans la super-classe au travers de laquelle est manipulé l'objet dérivé
  - une méthode redéfinie dans les sous-classes ne peut avoir un accès plus restrictif que la méthode d'origine
  - une méthode redéfinie dans les sous-classes ne peut lancer plus d'exceptions que la méthode d'origine



## méthodes redéfinies

- exemple:

```
public class A {
 public int display() { ... }
 public void affiche() { ... }
}

public class B extends A {
 public String getchaine() { ... }
 public void affiche() { ... }
}

public class C extends B {
 public String getchaine() { ... }
 public int display() { ... }
}
```



## méthodes redéfinies

- exemple (quelles sont les méthodes exécutées?):

```
public class Test {
 public static void main(String[] args) {
 B pb=new B();
 A pa=pb;
 B ppb=new C();
 A ppa=ppb;
 pa.affiche();
 pb.affiche();
 pa.display();
 pb.display();
 ppa.affiche();
 ppa.display();
 ppb.getchaine();
 ppb.getchaine();
 ppb.display();
 }
}
```



## méthodes redéfinies

- **exemple (solution):**

```
public class Test {
 public static void main(String[] args) {
 B pb=new B();
 A pa=pb;
 B ppb=new C();
 A ppa=ppb;
 pa.affiche(); // affiche() de B : polymorphisme
 pb.affiche(); // affiche() de B
 pa.display(); // display() de A
 pb.display(); // display() de A : héritage
 ppa.affiche(); // affiche() de B : polymorphisme
 ppa.display(); // display() de C : polymorphisme
 ppa.getchaine(); // ERREUR! getchaine()
 // n'existe pas dans A
 ppb.getchaine(); // getchaine() de C : polymorphisme
 ppb.display(); // display() de C : polymorphisme
 }
}
```



## annotation @Override

- L'annotation facultative **@Override** sur une méthode redéfinie permet au compilateur de détecter une anomalie dans la signature de la méthode

```
public class Personne {
 public void affiche () {...}
}
public class Employe extends Personne {
 @Override public void affiche() {...}
}
```

- le développeur manifeste son intention de redéfinir une méthode en y ajoutant l'annotation **@Override**
- le compilateur vérifie alors s'il existe bien une méthode de même signature dans les super-classes, et signale une erreur dans le cas contraire



## atelier

- Java24: mise en œuvre du polymorphisme pour obtention du même résultat que Java23



## les classes et méthodes abstraites

- le polymorphisme
- **les classes et méthodes abstraites** → :
  - les classes abstraites
  - les méthodes abstraites
- les interfaces
- les exceptions
- compléments
- outils jar et javadoc
- la bibliothèque standard
- les collections



## les classes abstraites

- une classe abstraite est une classe non instanciable
- une classe abstraite a uniquement pour rôle de généraliser d'autres classes en devenant ainsi leur classe parente
  - une classe abstraite correspond en général à un concept de notre esprit tout en n'étant pas nécessairement justifiée dans le cadre de la modélisation du monde réel

```
// il s'agit de modéliser un composant graphique
public abstract class Component{
 .
 .
 .
}
```



## les classes abstraites

- l'intérêt est de factoriser les caractéristiques (attributs) et comportement (méthodes) d'objets différents
- s'il n'est pas possible de créer des objets d'une classe abstraite, il est en revanche possible de définir des références sur des objets des sous-classes

```
Component c=new Component(); // ERREUR !
Component cx=new JButton(); // OK, si JButton
 // hérite de Component
```



## les méthodes abstraites

- les méthodes abstraites (*abstract*) sont des méthodes sans définition qui doivent être définies dans les sous-classes

```
abstract class Maclasse {

 public abstract void f(); // méthode sans définition
}
```

- une classe dont au moins une méthode est abstraite est nécessairement une classe abstraite
- toutes les méthodes abstraites doivent être redéfinies dans les sous-classes sous-peine d'obtenir des sous-classes abstraites.



## les interfaces

- le polymorphisme
- les classes et méthodes abstraites
- **les interfaces** →
  - définition
  - utilisation
  - intérêt
- les exceptions
- compléments
- outils jar et javadoc
- la bibliothèque standard
- les collections



## définition

- les classes de Java ne supportent que l'héritage simple: une sous-classe ne peut pas hériter simultanément de plusieurs classes, contrairement au C++ ou à Eiffel
- Java propose un autre concept autorisant une certaine forme d'héritage multiple par le biais d'interfaces
- une interface est une structure qui ne comportait, avant Java 8, que des méthodes abstraites



## définition

### ■ exemple:

```
public interface Carnivore {
 public abstract void devorer(Animal a);
}
```

- une interface ressemble à une classe abstraite dans son principe
- une interface peut hériter d'une autre interface, pas d'une classe
- une classe, même si elle hérite déjà d'une autre classe, peut implémenter une ou plusieurs interfaces
- les méthodes d'une interface sont toujours abstraites (plus en Java 8 !)
- les méthodes d'une interface sont implicitement publiques, et tout modificateur d'accès autre que **public** est interdit



## utilisation

- une interface ne peut être utilisée seule: elle nécessite une ou plusieurs classes qui auront à **l'implémenter**, en définissant obligatoirement les méthodes déclarées dans l'interface

```
// la classe Lion implémente Carnivore
public class Lion implements Carnivore {
 private int poids;
 private int age;
 private boolean sexe;
 .
 .
 .
 public void devorer(Animal x) {
 // redéfinition nécessaire ici
 }
}
```



## intérêt

- l'intérêt majeur pour une classe d'implémenter une interface est de permettre la manipulation des instances de cette classe via une référence du type de l'interface (manipulation d'un objet dérivé)

```
Carnivore c=new Lion();
c.devorer(new Gazelle());
```

- cela permet à des objets de types différents, mais ayant un comportement commun (celui déclaré dans l'interface), d'être considéré sous un même aspect, facilitant ainsi leur manipulation



## intérêt

- bien qu'essentiellement destinées à la manipulation du polymorphisme, les interfaces peuvent parfois contenir des données
  - les données définies dans une interface sont obligatoirement `public static final`, même si ces modificateurs sont absents
  - les constantes d'une application pourront avantageusement être définies dans une interface, surtout si elles sont utilisées par de nombreuses classes



## intérêt

- une interface peut hériter d'autres interfaces

```
public interface Omnivore extends Carnivore {
 . . .
}
```
- la classe qui implémente l'interface `Omnivore` doit redéfinir toutes les méthodes de cette interface et celles de `Carnivore`

## interfaces depuis Java 8

- depuis Java 8, les interfaces peuvent désormais comporter:
  - des méthodes par défaut, qui ne sont pas abstraites et seront héritées par les classes qui implémenteront l'interface
    - ces classes pourront redéfinir ces méthodes si elles le souhaitent
    - ces méthodes par défaut ne peuvent pas prendre le nom des méthodes de la classe `Object`
    - en cas d'héritage multiple avec des méthodes par défaut de même nom, la classe devra redéfinir la méthode
  - des méthodes déclarées static dont bénéficieront les classes qui implémenteront l'interface
    - ces méthodes `static` ne peuvent pas être redéfinies, car ce ne sont pas des méthodes d'instance

## exemple

### ■ exemple :

```
@FunctionalInterface
public interface Function<T,R> {
 R apply(T t); ← méthode abstraite
 default<V> Function<T,V> andThen(← ces méthodes (concrètes)
 seront héritées par les
 classes qui implémenteront
 l'interface
 Function<? super R,>? extends V> after){...}
 default<V> Function<V,R> composed(← cette méthode
 static sera héritée
 par les classes qui
 implémenteront
 l'interface
 Function<? super V,>? extends T> before){...}
 static <T> Function<T,T> identity(){...}
}
```

- les interfaces ressemblent donc désormais davantage à des classes abstraites en évitant le problème de l'héritage multiple en diamant (absence d'attributs)



## exemple (suite)

- **interface java.lang.Comparable**

```
public interface Comparable<T> {
 int compareTo(T o);
}
```

- on souhaite créer une interface nommée **Orderable** plus commode d'utilisation

```
public interface Orderable<T> extends Comparable<T> {
 public default boolean isAfter(T other) {
 return compareTo(other) > 0;
 }
 public default boolean isBefore(T other) {
 return compareTo(other) < 0;
 }
 public default boolean isSameAs(T other) {
 return compareTo(other) == 0;
 }
}
```



## exemple (suite)

- la classe **Personne** implémente l'interface **Orderable**

```
public class Personne implements Orderable<Personne> {
 private final String nom;

 public Personne(String nom) {
 this.nom = nom;
 }

 @Override
 public int compareTo(Personne p) {
 return nom.compareTo(p.nom);
 }

 // la classe hérite des méthodes par défaut de
 // l'interface Orderable
}
```



## exemple (suite)

- mise en oeuvre:

```
public class TestOrderable {
 public static void main(String[] args) {
 Personne martin= new Personne("MARTIN");
 Personne durand= new Personne("DURAND");

 System.out.println("comparaison: " +
 martin.compareTo(durand));
 System.out.println("Martin > Durand : " +
 martin.isAfter(durand));
 System.out.println("Martin < Durand : " +
 martin.isBefore(durand));
 System.out.println("Martin == Durand : " +
 martin.isSameAs(durand));
 }
}
```



## atelier

- Java25: utilisation d'un timer pour déclencher l'affichage d'un message



## les exceptions

- le polymorphisme
- les classes et méthodes abstraites
- les interfaces
- **les exceptions** →
  - principe et mots-clés
  - hiérarchie des classes d'exception
  - traitement sur place des exceptions
  - la classe java.lang.Exception
  - la clause finally
  - l'instruction try-with-resources
  - propagation des exceptions
  - lancer des exceptions
  - les exceptions personnalisées
  - les assertions
- compléments
- outils jar et javadoc
- la bibliothèque standard
- les collections



## principe et mots-clés

- les anomalies survenant lors de l'exécution d'un programme Java peuvent être traités par un mécanisme d'exceptions
- ce mécanisme est contrôlé par les mots-clés `try`, `catch`, `throw`, `throws` et `finally`
- une exception est une instance d'une sous-classe de la classe `java.lang.Throwable`
- une exception est un objet lancé par l'instruction `throw`



## hiérarchie des classes d'exception

- toutes les exceptions sont nécessairement des objets dérivés de `java.lang.Throwable`. La hiérarchie des exceptions est la suivante:

```
Throwable
 Error
 Exception
 IOException
 RuntimeException
```

- les exceptions dérivées de `Error` concernent la machine virtuelle et ne peuvent être gérées par le programmeur



## hiérarchie des classes d'exception

- les exceptions dérivées de `java.lang.Exception` concernent directement le programmeur:
  - celles dérivées de `IOException` sont générées par certaines méthodes d'entrées/sorties et doivent obligatoirement être traitées
  - celles dérivées de `RuntimeException` sont les conséquences d'erreurs de programmation (par exemple `IndexOutOfBoundsException` pour un débordement de tableau)



## hiérarchie des classes d'exception

- les exceptions dérivées de **Error** et **RunTimeException** sont dites non contrôlées (unchecked)
- toutes les autres sont dites contrôlées (checked)
- le compilateur rend obligatoire la prise en compte des exceptions contrôlées



## traitement sur place des exceptions

- le mot-clé **try** permet de définir un bloc d'instructions susceptible de lancer une ou plusieurs exceptions
- suivant immédiatement le bloc **try**, le mot-clé **catch**, suivi d'un argument, permet de capturer une exception d'un certain type
  - il peut y avoir plusieurs **catch** successifs après un même bloc **try**: ceci permet de capturer des exceptions différentes pouvant se produire dans les instructions du bloc **try**
  - le mot-clé **finally** permet de définir un bloc d'instructions exécutées dans tous les cas



## traitement sur place des exceptions

```
public class Demo {
 public static void main (String args[]) {
 . . .
 int x=getEntier();
 }
 public static int getEntier () {
 BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
 int essai=3, val=0;
 do {
 try {
 String str = br.readLine (); // throws IOException
 val = Integer.parseInt (str); // throws NumberFormatException
 } catch(IOException ioe) {
 System.out.println("erreur d'E/S"+ioe.getMessage());
 return 0;
 } catch (NumberFormatException nbe) {
 System.out.println ("Il faut saisir une valeur entière !");
 continue;
 } break;
 } while (--essai>0);
 return val;
 }
}
```



## traitement sur place des exceptions

- dans cet exemple, les deux instructions présentes dans le bloc **try** lancent chacune un objet exception différent, en cas de problème rencontré dans leur exécution

```
try {
 String str = br.readLine (); // throws IOException
 val = Integer.parseInt (str); // throws
 //
 NumberFormatException
}
```

- la méthode `readLine()` peut lancer un objet de la classe `IOException`
- la méthode `parseInt(String)` peut lancer un objet de la classe `NumberFormatException`



## traitement sur place des exceptions

- les gestionnaires d'exception **catch** permettent de capturer des instances de la classe indiquée entre parenthèses ou des sous-classes, et ainsi de préciser le traitement approprié

```
 } catch (IOException ioe) { // ioe est une référence
 // sur l'objet capturé
 System.out.println("erreur d'E/S "+ioe.getMessage());
 return 0;
 } catch (NumberFormatException nbe) {
 System.out.println ("Il faut une valeur entière !");
 continue;
 }
```

- l'ordre des catch est important: les exceptions les plus précises doivent se trouver au début



## traitement sur place des exceptions

- un bloc **catch** unique peut attraper plusieurs types d'exceptions

- cela permet de rendre le code plus compact

- exemple:

```
 try {
 . . .
 } catch (IOException | SQLException ex) {
 logger.log(ex);
 ex.printStackTrace();
 }
```

- les classes d'exceptions ne doivent pas avoir de relation d'héritage entre-elles



## la classe `java.lang.Exception`

- les exceptions sont nécessairement des instances de sous-classes de la classe `Throwable` dont la classe `Exception` est dérivée
- toute exception peut donc être capturée en la considérant comme un objet de type `Throwable` ou de préférence `Exception`:

```
catch (Exception t) {
 System.out.println(t.getMessage());
 . . .
}
```



## la classe `java.lang.Exception`

- toute classe d'exception hérite de la méthode `getMessage()` de la classe `Throwable`, permettant d'obtenir une chaîne de caractères représentative de l'exception survenue
- la méthode `printStackTrace()` peut également être appelée pour obtenir une trace de la pile avant la survenue de l'exception



## la clause finally

- une clause **finally** garantit que ses instructions seront exécutées dans tous les cas, qu'il y ait exception ou non
  - cela permet de libérer des ressources: fermeture de fichiers, de connexions réseau, de connexions aux bases de données, etc...



## la clause finally

- exemple:

```
InputStream in = null;
try {
 in = new FileInputStream("file.txt");
 int data = in.read();
} catch (IOException e) {
 System.out.println(e.getMessage());
} finally {
 try {
 if(in != null) in.close();
 } catch(IOException e) {
 System.out.println("Failed to close file");
 }
}
```



## l'instruction try-with-resources

- l'instruction **try-with-resources** permet d'éviter l'usage du **finally**
  - une ressource est un objet qui doit être fermé (`close`) lorsqu'il n'est plus utilisé
  - l'instruction **try-with-resources** garantit que chaque ressource sera fermée à la fin du **try**
    - si une exception se produit lors de la création de la ressource, un `catch` est immédiatement exécuté
    - si une exception se produit dans le corps du `try`, toutes les ressources seront fermées avant l'exécution du `catch`
    - si une exception se produit lors de la fermeture de la ressource, elle sera annulée



## l'instruction try-with-resources

- tout objet qui implémente l'interface `java.lang.AutoCloseable` peut être utilisé comme ressource
  - sa méthode `close()` est alors automatiquement appellée

```
interface AutoCloseable{ public void close() throws Exception;}
```
  - ceci inclut également tous les objets qui implémentent `java.io.Closeable`



## l'instruction try-with-resources

- **exemple:**

```
static String readFirstLineFromFile(String path)
 throws IOException {
 try (BufferedReader br =
 new BufferedReader(new FileReader(path))) {
 return br.readLine();
 }
}
```



## atelier

- Java26: gestion sur place d'exceptions d'entrées-sorties



## propagation des exceptions

- le mécanisme consistant à utiliser les mots-clés `try` et `catch` dans la méthode où se produit l'exception est appelé traitement sur place
- il est possible d'adopter une autre stratégie de traitement en propageant l'objet exception
- c'est alors la méthode appelante de celle qui propage l'exception qui doit la prendre en compte, soit en la traitant sur place, soit en la propageant à nouveau



## propagation des exceptions

```
public class Demo {
 public static void main (String args[]) {
 . . .
 try {
 int x=getEntier();
 } catch(IOException ioe) {
 System.out.println("erreur d'E/S "+ioe.getMessage());
 } catch (NumberFormatException nbe) {
 System.out.println ("Il faut saisir une valeur entière !");
 }
 }
 public static int getEntier () throws IOException {
 BufferedReader br=new BufferedReader(new
InputStreamReader(System.in));
 int essai=3, val=0;
 String str = br.readLine (); // throws IOException
 val = Integer.parseInt (str); // throws NumberFormatException
 return val;
 }
}
```



## propagation des exceptions

- la méthode `getEntier()` ne traite pas les exceptions mais les propage
- la méthode appelante, ici `main`, se voit contrainte de traiter les exceptions lancées par `getEntier()`
- une méthode qui lance une exception **contrôlée** sans la traiter **doit obligatoirement le spécifier** dans sa déclaration par le mot-clé `throws`

```
public static void getEntier () throws IOException
{ . . . }
```

- seules les exceptions contrôlées ont obligatoirement à être mentionnées dans la clause `throws`
- une méthode redéfinie (polymorphe) ne peut pas lancer plus d'exceptions que la méthode qu'elle redéfinit



## lancer des exceptions

- une exception peut être lancée avec le mot-clé `throw`

```
if (...) {
 throw new IOException("acces impossible");
}
```

- si cette instruction ne figure pas dans un `try...catch`, ou si aucun `catch` n'est capable de la traiter, la méthode doit indiquer qu'elle la propage avec `throws`



## lancer des exceptions

- une exception peut être attrapée puis relancée

```
public static int readByteFromFile()
 throws IOException {
 try (InputStream in =
 new FileInputStream("a.txt")) {
 System.out.println("File open");
 return in.read();
 } catch (IOException e) {
 e.printStackTrace();
 throw e;
 }
}
```



## atelier

- Java27: propagation d'exceptions d'entrées-sorties



## les exceptions personnalisées

- dans certains cas, il peut être judicieux de définir sa propre classe d'exception
- dans cet exemple, **Depassement** est une sous-classe de la classe **Exception**:

```
public class Depassement extends Exception {
 public Depassement (String msg) {
 super (msg);
 }
}
```



## les exceptions personnalisées

- une méthode peut alors lancer une instance de cette classe par l'instruction **throw**

```
if (valeur > 1000) {
 throw new Depassement (valeur+ " trop grande");
}
```

- la chaîne de caractères transmise en argument du constructeur pourra être récupérée par la méthode **getMessage()** appliquée à l'objet exception capturé



## translation d'exceptions

- il est possible de cacher le type de l'exception produite en l'enveloppant dans une autre (wrapper exception)

```
public class DAOException extends RuntimeException
{
 public DAOException(Throwable cause) {
 super(cause);
 }
 public DAOException(String message, Throwable cause) {
 super(message, cause);
 }
}
```



## translation d'exceptions

- cela permet d'effectuer une translation d'exception, utile pour transformer une exception contrôlée en exception non contrôlée:

- dans la couche DAO:

```
public Personne findById(String id)
 throws DAOException {
 try {
 ...
 } catch (SQLException e) {
 throw new DAOException(
 "Personne non trouvée", e);
 }
}
```



## translation d'exceptions

- la classe **Throwable** contient la méthode **getCause()** qui permet d'obtenir l'exception enveloppée:
  - dans la couche service:

```
public Personne findById(String id) {
 ...
 } catch (DAOException e) {
 log.info("Personne non trouvée: " +
 e.getCause());
 }
}
```



## les assertions

- les assertions permettent au développeur de tester si des conditions sont vérifiées à des endroits précis dans une application
  - si une assertion n'est pas vérifiée, l'application s'arrête en affichant des informations sur l'assertion en cause
  - les assertions peuvent être validées ou non au lancement de l'application
- elles peuvent être utilisées dans une méthode pour vérifier:
  - les invariants internes
  - les invariants de code
  - les invariants de classe et de postconditions



## les assertions

- les assertions ne doivent pas être utilisées:
  - pour vérifier les arguments d'une méthode déclarée **public**
  - si des effets de bords peuvent survenir avec l'assertion



## les assertions

- syntaxe:

```
assert <boolean_expression> ;
assert <boolean_expression> : <detail_expression> ;
```

  - si l'expression booléenne est évaluée à **false**, une exception **java.lang.AssertionError** est lancée
  - le deuxième argument est converti en **String** et permet d'ajouter une description qui complète le message standard
- exemple:

```
int i = -1;
if(i<0){
 i = -i;
}
assert (i>=0) : "impossible "+i;
```



## les assertions

- les assertions sont invalidées par défaut
  - quand les assertions sont invalidées, l'application s'exécute aussi rapidement que si les assertions étaient absentes
- pour valider les assertions, utiliser l'option **-ea** de la JVM

```
java -enableassertions Appli
java -ea Appli
```

- on peut également préciser que les assertions ne sont validées que pour une classe ou un package donné

```
-ea:<nom de classe>
-ea:<nom de package>
```

- pour plus d'informations, consulter:

<http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html>



## atelier

- Java28: création d'exceptions personnalisées



## compléments

- le polymorphisme
- les classes et méthodes abstraites
- les interfaces
- les exceptions
- **compléments** 
- outils jar et javadoc
- la bibliothèque standard
- les collections
- méthodes à nombre variable d'arguments
- annotations
- la généricité



## méthodes à nombre variable d'arguments

- une méthode peut indiquer qu'elle peut recevoir un nombre variable d'arguments
  - public void methode(int val, **String...** mes);
  - cette méthode attend un premier paramètre de type **int**, et un nombre quelconque de paramètres de type **String**, y compris aucun
  - seuls les derniers arguments peuvent l'être en nombre variable
- lors de l'appel de cette méthode, il suffit d'indiquer les arguments qui lui sont transmis
  - methode(5, "alpha", "beta");



## méthodes à nombre variable d'arguments

- les arguments sont reçus dans un tableau qu'il suffit de parcourir:

```
public class Calcule {
 public static double somme(double... oper) {
 double somme=0;
 for(double val:oper){somme+=val;}
 return somme;
 }
 public static void main(String[] args){
 double s=Calcule.somme(45.7, 0.3, 18, 33);
 System.out.println("somme: "+s);
 }
}
```



## annotations

- les annotations, appelées aussi méta-données, permettent de marquer certaines classes, interfaces ou méthodes afin de leur ajouter des propriétés particulières
  - ces annotations peuvent ensuite être exploitées à la compilation ou à l'exécution pour automatiser certaines tâches
- la spécification Java EE et les frameworks (Hibernate, JSF, Spring) utilisent abondamment les annotations
  - Java SE propose des annotations standard, mais il est facile de développer ses propres annotations



## annotations standard

- Java propose trois annotations standard:

**@Deprecated**: permet de signaler une méthode comme obsolète

**@Override**: permet d'indiquer une méthode redéfinie

**@SuppressWarnings**: permet de supprimer des avertissements

- exemples:

```
public class Maclasse {
 . . .
 @Deprecated public int getDuree () {
 return duree; }
 @Override public String toString(){
 return super.toString();}
 @SuppressWarnings("deprecation")
 public int calcule(int x1,int x2){...}
}
```



## la généricité

- la généricité est un mécanisme qui permet de créer des classes, interfaces ou méthodes dont certains types sont inconnus, et déterminés ultérieurement à la compilation
  - cela permet d'éviter la création d'innombrables versions de la classe, de l'interface ou de la méthode
  - les paramètres génériques correspondent obligatoirement à des classes, interfaces ou énumération
    - les types primitif ne sont pas supportés (mais les classes enveloppes le sont...)



## la généricité

- exemple: classe générique

```
public final class Optional<T>{
 ...
 public T get();
 public T orElse(T other);
 ...
}
```

- T est ici un paramètre générique qui sera substitué par un type réel à l'utilisation de la classe **Optional**

```
Optional<Personne> op = dao.findById(1675456);
if(op.isPresent()) {
 Personne p = op.get();
 ...
}
```



## la généricité

- exemple: interface générique

```
public interface Comparable<T>{
 int compareTo(T o);
}
```

- T est ici un paramètre générique qui sera substitué par un type réel à l'utilisation de l'interface **Comparable**

```
public class Personne implements Comparable<Personne> {
 private String nom;
 private String prenom;
 private int age;

 public int compareTo(Personne o){
 return nom.compareTo(p.nom);
 }
 . . .
```



## la généricité

- exemple: méthode générique

```
public class Arrays{
 public static <T> Stream<T> stream(T[] array);
}
```

- **T** est ici un paramètre générique qui sera substitué par un type réel à l'utilisation de la méthode **stream**
- **<T>** devant le type de retour de la méthode, indique au compilateur que **T** est un paramètre générique

```
String[] mots = {"alpha", "beta", "gamma"};
Stream<String> st = Arrays.stream(mots);
```



## la généricité

- les méthodes peuvent recevoir des arguments de type générique et en retourner

```
boolean containsAll(Collection<?> c)
```

- cette méthode accepte toute collection, peu importe le type d'objets qu'elle contient

```
void forEach(Consumer<? super T> action)
```

- cette méthode accepte tout objet **Consumer**, dont le paramètre générique est de type **T** ou super-type de **T**

```
static <T> Stream<T> generate(Supplier<? extends T> s)
```

- cette méthode accepte tout objet **Supplier**, dont le paramètre générique est de type **T** ou sous-type de **T**



## la généricité

- les paramètres génériques transmis doivent correspondre exactement au type attendu

```
void methode1(Collection<Object> c)
```

- cette méthode accepte uniquement une collection d' **Object**
- une collection de type **Collection<Personne>** est refusée par le compilateur

```
void methode2(Collection<Personne> action)
```

- cette méthode accepte uniquement une collection de **Personne**
- une collection de type **Collection<Employe>** est refusée par le compilateur, même si la classe **Employe** hérite de la classe **Personne**



## les outils jar et javadoc

- le polymorphisme
- les classes et méthodes abstraites
- les interfaces
- les exceptions
- compléments
- **outils jar et javadoc** →
  - Jar
  - Javadoc
- la bibliothèque standard
- les collections



## jar

- **jar** est un outil compris dans le JDK permettant d'archiver des fichiers de tous types en les comprimant au format zip
- l'intérêt principal d'une archive au format **jar** est sa commodité d'utilisation
- les fichiers constituant un programme Java, application, applet, servlet, JavaBean seront avantageusement archivés à l'aide de cet outil
- le nom d'une archive au format **jar** peut être quelconque, avec ou sans extension, mais l'extension **.jar** permet de l'identifier



## jar

- la commande **jar** comporte 3 options principales:
  - **c** pour créer une archive
  - **x** pour extraire le contenu d'une archive
  - **t** pour lister le contenu d'une archive
- l'option **v** (verbose) ajoute des indications lors de l'exécution d'une opération
- l'option **f** permet d'indiquer le nom du fichier archive
- l'option **m** permet de modifier le fichier **manifest** par défaut



## jar

### ■ exemples:

- archivage des fichiers Alpha.class, Beta.class dans le fichier alpha.jar:

```
jar cvf alpha.jar Alpha.class Beta.class
```

- archivage du contenu du répertoire rep dans l'archive gestion.jar:

```
jar cvf gestion.jar rep
```

- affichage du contenu de l'archive gestion.jar:

```
jar tvf gestion.jar
```

- extraction du contenu de l'archive gestion.jar:

```
jar xvf gestion.jar
```



## jar

- pour créer correctement un fichier jar d'application, il faut empaqueter les répertoires correspondant aux packages, pas seulement les fichiers .class
- lorsqu'une application est empaquetée dans un fichier jar, il suffit de préciser ce fichier dans le classpath pour permettre son exécution:

```
java -classpath appli.jar gestion.Appli
```



## jar

- toute archive créée à l'aide de `jar` comporte un fichier appelé fichier **MANIFEST.MF** qui permet de décrire le contenu de l'archive
  - il s'agit d'un fichier texte dont le contenu est exploité par certains programmes dont `java`, notamment
- le contenu de ce fichier peut être modifié par l'option `m` qui permet de préciser le nom d'un fichier texte, contenant certaines indications:

```
jar cvfm appli.jar monmanifest *.class
```

- contenu du fichier monmanifest:  
`Main-class: Appli`
- alors, l'application peut être lancée par la commande:  
`java -jar appli.jar`
- ou un double clic sur le fichier `appli.jar`



## atelier

- Java29: création d'un fichier jar exécutable à partir de l'application du Java24



## javadoc

- **javadoc** est un utilitaire fourni gratuitement par Oracle, permettant de créer des fichiers de documentation au format HTML à partir de fichiers sources java
- il extrait des fichiers sources les noms des classes et/ou interfaces, les noms et types des membres protected ou public
- il extrait également certains commentaires ainsi que certaines balises propres à l'outil **javadoc**



## javadoc

- **syntaxe:**

```
javadoc [options] [nomsdepackage] [fichierssources]
```

- **options:**

- sourcepath: chemin d'accès aux fichiers

- d : répertoire de destination des fichiers HTML

- **nomsdepackage:**

- Suite de noms de packages, séparés par des espaces. Il faut indiquer tous les packages à extraire, y compris les sous-packages.

- **fichierssources :**

- Suite de noms de fichiers, séparés par des espaces . Ceux-ci peuvent comporter des astérisques.

- **exemple:**

```
javadoc -d C:\home\doc
C:\home\src\java\awt\Button.java
```



## javadoc

- des commentaires spécifiques à l'outil Javadoc peuvent être avantageusement insérés dans les fichiers sources Java:

```
/**
 *
 */

■ exemple:
/**
 * Returns the character at the specified index. An index
 * ranges from <code>0</code> to <code>length() - 1</code>.
 *
 * @param index the index of the desired character.
 * @return the desired character.
 * @see java.lang.Character#charValue()
 */
public char charAt(int index) { ... }
```



## javadoc

- ces commentaires sont à placer avant chaque classe ou interface, ainsi qu'avant chaque méthode que l'on souhaite documenter
- les commentaires Javadoc peuvent également comporter des balises HTML, par exemple:

```
/**
 * Ce mot est en gras
 *
 */
```

- cependant, certaines balises comme <H1>, ou plus généralement des balises de mise en page de document ne devraient pas être utilisées, Javadoc assurant de lui-même la mise-en-page



## javadoc

- les balises **Javadoc** sont des mots-clés précédés du signe @ et insérés dans des commentaires **Javadoc**, entraînant une présentation spécifique à la balise
  - @author
  - @deprecated
  - @exception,@throws
  - {@link}
  - @param
  - @return
  - @see
  - @serial
  - @serialData
  - @serialField
  - @since
  - @version



## javadoc

- **principales balises:**
  - @author : permet de préciser le nom de l'auteur du programme
  - @deprecated: pour indiquer que la méthode documentée ne doit plus être utilisée
  - @exception,@throws: pour préciser les classes d'exceptions lancées par la méthode documentée
  - {@link}: pour insérer un lien hypertexte dans le commentaire
  - @param: pour indiquer les arguments d'une méthodes
  - @return: pour indiquer les données rentrées par une méthode
  - @see: définit un lien hypertexte dans une rubrique « See also »
  - @serial: documente les champs sérialisables
  - @since: pour indiquer une version ou une date de validité
  - @version : pour indiquer la version du programme



## atelier

- démonstration: ajout de commentaires Javadoc à l'application du projet Java24 puis génération de la javadoc



## la bibliothèque standard

- le polymorphisme
- les classes et méthodes abstraites
- les interfaces
- les exceptions
- compléments
- outils jar et javadoc
- **la bibliothèque standard** →
  - les principaux packages
  - la classe Object
  - la classe System
  - la classes Math
  - les classes enveloppes
- les collections



## les principaux packages

- la bibliothèque est un ensemble de classes fournies avec le JDK, ce qui soulage le programmeur de la définition et de la mise-au-point de ces classes, parfois très complexes
- les classes se trouvent dans des sous-packages de java et javax :
  - lang, util, io, net, awt, applet, beans, rmi, sql, security, math, text
  - swing
- ces packages contiennent à leur tour d'autres packages



## les principaux packages

- **java.lang :**  
les classes de ce package ont un rôle privilégié puisqu'elles concernent directement le langage. On y trouve notamment les classes Object, Class, System, Thread, String, Number, Integer, Long, Character, Boolean, Float, Double, Math.  
il n'est pas nécessaire d'importer les classes de ce package pour pouvoir les utiliser
  - la classe Object est particulière car elle est implicitement classe de base de toute classe, que celle-ci soit créée par le programmeur ou qu'elle fasse partie de la bibliothèque.



## les principaux packages

- **java.util:**

ce package contient des classes utilitaires, comme `Vector`, `Date`, `Calendar`, `Dictionary`, `Random`

- **java.io:**

ces classes concernent les entrées/sorties de tous type, y compris sur des fichiers. les classes `File`, `RandomAccessFile`, `InputStream`, `OutputStream`, `ObjectInputStream`, `ObjectOutputStream` en font partie



## les principaux packages

- **java.net:**

ces classes permettent l'accès à un réseau. on y trouve notamment `Socket`, `ServerSocket`, `URL`, `URLConnection`

- **java.awt:**

les classes de ce package sont les plus nombreuses. Elles permettent de créer des interfaces graphiques utilisateur. On y trouve par exemple `Color`, `Font`, `Window`, `Frame`, `Menu`, `Button`, `Choice`, `Checkbox`, `Dialog`, `FileDialog`, `TextField`



## les principaux packages

- **java.applet:**

ces classes permettent de réaliser des applets. La classe **Applet** en fait partie

- **java.beans:**

les classes de ce package permettent de créer des Java beans ou des outils pour utiliser ou les créer. Citons les classes **Beans**, **Introspector**, **FeatureDescriptor**



## les principaux packages

- **java.rmi:**

les classes de ce package permettent d'accéder côté client à des objets distribués sur des serveurs. Les classes **Naming**, **RMISecurityManager** et l'interface **Remote** en font partie

- **java.sql:**

les classes de ce package permettent l'accès aux bases de données. **DriverManager**, **Types**, **Dta**, **Time**, **Timestamp** en font partie



## les principaux packages

- **java.security:**

les classes de ce package concernent les fonctions de sécurité de Java. **Security**, **Signature**, **Identity** en font partie.

- **java.math:**

contient seulement les classes **BigDecimal** et **BigInteger** permettant de dépasser les limites des types de base



## les principaux packages

- **java.text:**

contient les classes permettant de rendre les programmes Java indépendants de la situation géographique et de la langue. Citons les classes **Format**, **DateFormat**, **BreakIterator**, **Collator**

- **javax.swing:**

ce package contient les classes dites « swing » permettant de construire des interfaces graphiques modernes. On y trouve par exemple **JFrame**, **JButton**, **JCheckbox**, **JTextField**



## la classe Object

- cette classe est la classe de base de toute classe, même si le mot-clé `extends` n'est pas précisé
- elle comporte des méthodes qui seront héritées par tout objet, ce qui donne à ce dernier un comportement par défaut:

```
public class Object {
 public Object();
 public boolean equals(Object obj);
 public final Class getClass();
 public int hashCode();
 public String toString();
 protected Object clone();
}
```



## la classe Object

- la méthode `getClass` retourne un objet `Class`, permettant ainsi d'obtenir quantité d'informations sur la classe d'un objet
- la méthode `clone` permet de dupliquer un objet par copie membre à membre des champs
- la méthode `equals` permet la comparaison des références de deux objets. Cette méthode est souvent redéfinie dans les classes afin de permettre la comparaison d'objets
- la méthode `hashCode` retourne un code numérique pouvant servir de clé dans des structures de données telles que les `HashTable`



## la classe Object

- lorsqu'une classe redéfinit la méthode **equals**, elle doit également redéfinir la méthode **hashcode**
- la méthode **hashcode** doit retourner un entier dont la valeur peut être obtenue arbitrairement, de préférence par un calcul faisant intervenir les attributs
- deux objets de même état doivent retourner la même valeur de hashcode



## la classe Object

```
public class Personne{
 private String nom;
 private String prenom;
 private int age;

 public boolean equals(Object obj){
 if(! obj instanceof Personne) return false;
 Personne p=(Personne)obj;
 if(nom.equals(p.nom) && prenom.equals(p.prenom) &&
 age==p.age) return true;
 }
 public int hashCode (){
 return super.hashCode ()+7*nom.hashCode () +
 11*prenom.hashCode ()+13*age;
 }
}
```



## la classe Object

- la méthode `toString` permet d'obtenir la représentation d'un objet quelconque sous la forme d'une chaîne de caractères. Cette méthode est notamment appelée par la méthode `println(Object)`
- il est judicieux de redéfinir la méthode `toString` dans toute nouvelle classe, permettant ainsi d'obtenir une représentation la plus complète possible d'une instance de cette classe



## la classe Object

```
public class Compteur {
 . . .
 public String toString() { // méthode redéfinie
 return "valeur "+cpt;
 }
}
public class TstCounter {
 public static void main(String[] args) {
 Compteur c1=new Compteur (5);
 System.out.println(c1); // valeur: 5
 }
}
```



## la classe Object

- outre ses méthodes, l'intérêt de la classe `Object` est de permettre la manipulation de tout objet via une référence du type `Object`, en argument en en retour de méthode
- par exemple, la classe `java.util.Vector` permet de stocker des objets dans une collection:

```
public class Vector {
 public Vector();
 public Vector(int capacite_initiale);
 public boolean add(Object o);
 public Object get(int index);
 public int size();
 public void remove(int index);
}
```



## la classe Object

```
public class MaClass {
 public static void main(String[] args) {
 Compteur c1=new Compteur (7);
 Vector v1=new Vector(10);
 v1.add(c1);
 . . .
 Compteur c=(Compteur)v1.get(0);
 }
}
```

- la contrepartie de cette manipulation réside dans la quasi-nécessité d'effectuer, à un moment ou à un autre, le transtypage de l'objet dans sa classe d'origine:

```
Compteur c=(Compteur)v1.get(0);
```



## atelier

- Java30: redéfinition de la méthode `toString()` dans les classes du projet Java24



## la classe System

- il s'agit de l'une des classes les plus utilisées, à cause de son champ `out`, utilisé dans:

`System.out.println()`

- elle ne comporte que des champs et des méthodes de classe, et ne peut être instanciée

- elle comporte 3 champs publics, `in`, `out`, `err`, qui sont des objets désignant respectivement:

in : le fichier standard d'entrée (le clavier par défaut)  
out : le fichier standard de sortie (l'écran par défaut)  
err : le fichier standard d'erreur (l'écran par défaut)

- ces fichiers peuvent être redirigés vers d'autres, au moyen des méthodes: `setIN`, `setOut`, `setErr`



## la classe System

```
public final class System {
 public static PrintStream out;
 public static PrintStream err;
 public static InputStream in;

 public static void exit(int status);
 public static void gc();
 public static Map<String, String> getenv();
 public static String getenv(String name);
 public static Properties getProperties();
 public static String getProperty(String key);
 public static long currentTimeMillis();
 public static void arraycopy(Object src, int srcPos, Object dest,
 int destPos, int length);
 public static void setOut(PrintStream out);
 public static void setErr(PrintStream err);
 public static void setIn(InputStream in);
 . . .
}
```



## la classe System

- parmi les méthodes les plus utilisées de cette classe, citons:
  - `exit`, qui permet de mettre fin à une application, en renvoyant un code de retour au système  
`System.exit(0);`
  - `gc`, qui permet de solliciter l'intervention du garbage collector de façon à libérer la mémoire occupée par des objets non référencés  
`System.gc();`
  - `getenv`, qui permet d'obtenir la liste des variables d'environnement sous la forme d'une collection Map.  
`Map<String, String> env=System.getenv();`



## la classe System

- `getProperties`, qui permet d'obtenir la liste des propriétés du système sous la forme d'une collection de type `Properties`

```
Properties props=System.getProperties();
```

- sa variante surchargée permet d'obtenir la valeur d'une propriété dont le nom est passée en argument:

```
String separateur=
 System.getProperty("file.separator");
```

- `currentTimeMillis`, qui permet d'obtenir la date courante exprimée en millisecondes. La différence entre deux appels indique le temps d'exécution

```
long t0=System.currentTimeMillis();
```

- `arraycopy`, qui permet de copier efficacement des tableaux: `arraycopy(tab1, 0, tab2, 0, tab1.length);`



## travaux pratiques

- Java31: déterminer le temps d'exécution d'une méthode dans une application



## la classe Runtime

- cette classe permet à une application d'obtenir un objet `Runtime` lui donnant accès à son environnement

```
public final class Runtime {
 public static Runtime getRuntime();
 public long freeMemory();
 public long maxMemory();
 public long totalMemory();
 public void load(String filename);
 public void loadLibrary(String libname);
 public Process exec(String command);
 public Process exec(String[] cmdarray);
 . . .
}
```



## la classe Runtime

- la méthode `getRuntime` permet à une application d'obtenir une instance de `Runtime`

```
Runtime rt=Runtime.getRuntime();
```

- `freeMemory`, `maxMemory`, `totalMemory` qui fournissent, en octets, les quantité de mémoire libre, maximum et totale de la VM
- `load`, permet de charger une bibliothèque dynamique, dont le nom de fichier est fourni en argument

```
rt.load("/home/lib/libX11.so");
```

- sa variante `loadLibrary` permet de charger une bibliothèque dynamique dont le nom est fourni en argument:

```
rt.load("libX11.so");
```



## la classe Runtime

- `exec`, permet d'exécuter dans un processus distinct la commande dont le nom est transmis en argument

```
rt.exec("winword.exe");
```

- ses variantes surchargées autorisent le passage d'arguments au lancement de la commande, la transmission de variables d'environnement, la transmission du répertoire de travail



## la classe Math

- la classe `java.lang.Math` propose des méthodes de classes permettant d'effectuer de nombreux calculs, principalement sur des données de type `double`, comme:
  - des opérations trigonométriques et hyperboliques
  - des calculs d'arrondis
  - des racines carrées et cubiques
  - une fonction aléatoire
- elle propose également les valeur `pi` et `e` (base des logarithmes népériens), via les constantes de classe respectives `PI` et `E`



## la classe Math

```
public class Math {
 public static double abs(double a); // valeur abs sur double
 public static float abs(float a); // valeur absolue sur float
 public static int abs(int a); // valeur absolue sur int
 public static long abs(long a); // valeur absolue sur long
 public static double acos(double a); // arc cosinus
 public static double asin(double a); // arc sinus
 public static double atan(double a); // arc tangente
 public static double atan2(double y,double x); // coordonnées polaires
 public static double cbrt(double a); // racine cubique
 public static double ceil(double a); // arrondi entier supérieur
 public static double cos(double a); // cosinus
 public static double cosh(double a); // cosinus hyperbolique
 public static double exp(double a); // exponentielle
 public static double expm1(double a); // (exp x) -1
```



## la classe Math

```
public static double floor(double a); // arrondi entier inférieur
public static double hypot(double x,double y); // hypothénuse
public static double IEEEremainder(double f1,double f2); // reste IEEE
f1/f2
public static double log(double a); // logarithme népérien
public static double log10(double a); // logarithme décimal
public static double log1p(double x); // log (1+x)
public static double max(double a,double b); // maximum de a et b
public static float max(float a,float b); // maximum de a et b
public static int max(int a,int b); // maximum de a et b
public static long max(long a,long b); // maximum de a et b
public static double min(double a,double b); // minimum de a et b
public static float min(float a,float b); // minimum de a et b
public static int min(int a,int b); // minimum de a et b
public static long min(long a,long b); // minimum de a et b
```



## la classe Math

```
public static double pow(double a,double b); // a puissance b
public static double random(); // valeur aléatoire
public static double rint(double a); // arrondi entier
public static long round(double a); // arrondi entier long
public static int round(float a); // arrondi entier
public static float signum(float f); // signe
public static double sin(double a); // sinus
public static double sinh(double x); // sinus hyperbolique
public static double sqrt(double a); // racine carrée
public static double tan(double a); // tangente
public static double tanh(double x); // tangente hyperbolique
public static double toDegrees(double angrad); // conversion en degrés
public static double toRadians(double angdeg); // conversion en radians
public static double ulp(double d); // ulp
public static float ulp(float f); // ulp
}
```



## les classes enveloppes

- il est parfois nécessaire de manipuler des objets plutôt que des données de type primitif, par exemple avec les collections
- afin de faciliter ces manipulations, la bibliothèque Java propose des classes dites "enveloppes" (wrapper classes) qui encapsulent chacun des types primitifs de Java:
  - java.lang.Boolean
  - java.lang.Short
  - java.lang.Integer
  - java.lang.Long
  - java.lang.Character
  - java.lang.Float
  - java.lang.Double



## les classes enveloppes

- exemple avec la classe `java.lang.Integer`:

```
public class Integer
{
 public Integer(int value);
 public Integer(String s);
 public int intValue();
 public static int parseInt(String s);
 public static int parseInt(String s, int radix);
 public static Integer valueOf(int i);
 public static Integer valueOf(String s);

 .
 .
}
```



## les classes enveloppes

```
Integer x=new Integer(6); // construction
 // d'un objet Integer
.
.
int v=x.intValue(); // on récupère la
 // valeur de cet objet
```

- la méthode `parseInt` est très utile, car elle permet d'obtenir un entier à partir d'une chaîne de caractères représentant un nombre entier

```
String s="123";
int i=Integer.parseInt(s); // valeur 123
```

- de façon similaire, il existe les méthodes `parseBoolean`, `parseByte`, `parseShort`, `parseLong`, `parseFloat`, `parseDouble` dans les autres classes enveloppes



## la classe Date

- la classe `java.util.Date` permet de représenter une date en années, mois, jours, heures, minutes, secondes en s'appuyant sur le nombre de millisecondes écoulées depuis le 01/01/1970
  - elle comporte aujourd'hui peu de méthodes utiles, la plupart étant obsolètes

```
public class Date {
 public Date(); // représente la date du jour
 public Date(long date); // date associée au nombre de ms
 public boolean after(Date when); // vrai si date postérieur à when
 public boolean before(Date when); // vrai si date antérieur à when
 public Object clone();
 public int compareTo(Date anotherDate);
 public boolean equals(Object obj);
 public int hashCode();
 public String toString(); // conversion en chaîne
 public void setTime(long time); // fixe la date à partir de time
 public long getTime(); // fournit le nombre de ms associé
}
```



## la classe DateFormat

- l'affichage ou la saisie de dates nécessite souvent de faire appel à la classe abstraite

`java.text.DateFormat`

```
public abstract class DateFormat {
 public static final DateFormat getInstance();
 public static final DateFormat getDateInstance();
 public static final DateFormat getDateInstance(int style);
 public static final DateFormat getDateInstance(int style, Locale aLocale);
 public static final DateFormat getTimeInstance();
 public static final DateFormat getTimeInstance(int style);
 public static final DateFormat getTimeInstance(int style, Locale aLocale);
 public static final DateFormat getDateTimeInstance();
 public static final DateFormat getDateTimeInstance(int dateStyle,
 int timeStyle);
 public static final DateFormat getDateTimeInstance(int dateStyle,
 int timeStyle, Locale aLocale);
 public Calendar getCalendar();
 public void setCalendar(Calendar newCalendar);
 public Date parse(String source) throws ParseException;
}
```



## la classe DateFormat

- la classe DateFormat étant abstraite, il n'est pas possible de l'instancier: il faut faire appel à l'une de ses méthodes de classe `getXXXXInstance`

- exemple:

```
Date d=new Date();
System.out.println("Date: "+d);

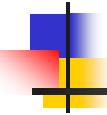
DateFormat df=
 DateFormat.getDateInstance(DateFormat.LONG, Locale.FRANCE);
System.out.println("Date en français: "+df.format(d));
■ le style peut être LONG, MEDIUM ou SHORT
■ l'objet Locale désigne le pays et/ou la langue
```



## la classe GregorianCalendar

- la classe `java.util.GregorianCalendar` est la plus utilisée dans la manipulation de dates

```
public abstract class GregorianCalendar extends Calendar {
 public GregorianCalendar() ;
 public GregorianCalendar(Locale aLocale) ;
 public GregorianCalendar(int year, int month, int dayOfMonth) ;
 public GregorianCalendar(int year, int month, int dayOfMonth,
 int hourOfDay, int minute, int second) ;
 public final Date getTime() ;
 public final void setTime(Date date) ;
 public long getTimeInMillis() ;
 public void setTimeInMillis(long millis) ;
 public int get(int field) ;
 public abstract void add(int field, int amount) ;
 public String toString() ;
 public void set(int field, int value) ;
 public final void set(int year, int month, int date) ;
 public final void set(int year, int month, int date, int hourOfDay,
 int minute, int second) ;
}
```

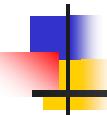


## la classe GregorianCalendar

- exemple:

```
GregorianCalendar g=new GregorianCalendar();
String [] mois_annee=
{"Janvier","Février","Mars","Avril","Mai","Juin","Juillet",
"Aout","Septembre","Octobre","Novembre","Décembre"};

int jour=g.get(GregorianCalendar.DAY_OF_MONTH);
int mois=g.get(GregorianCalendar.MONTH);
int annee=g.get(GregorianCalendar.YEAR);
int heure=g.get(GregorianCalendar.HOUR_OF_DAY);
int mn=g.get(GregorianCalendar.MINUTE);
int sec=g.get(GregorianCalendar.SECOND);
System.out.println("Nous sommes le: " +jour+
+mois_annee[mois]+" "+annee);
System.out.println("Il est: "+heure+"h "+mn+" mn "+sec+" sec");
```



## atelier

- Java32: afficher la date du jour et l'heure en français



## les collections

- le polymorphisme
  - les classes et méthodes abstraites
  - les interfaces
  - les exceptions
  - compléments
  - outils jar et javadoc
  - la bibliothèque standard
  - **les collections** →
- présentation
  - l'interface Collection
  - interfaces Set, List, Queue et Map
  - classes d'implémentation
  - collections et généricité
  - parcours de collections
  - comparaison d'objets
  - collections de type List
  - collections de type Set
  - collections de type Queue
  - collections de type Map
  - collections et multi-threading



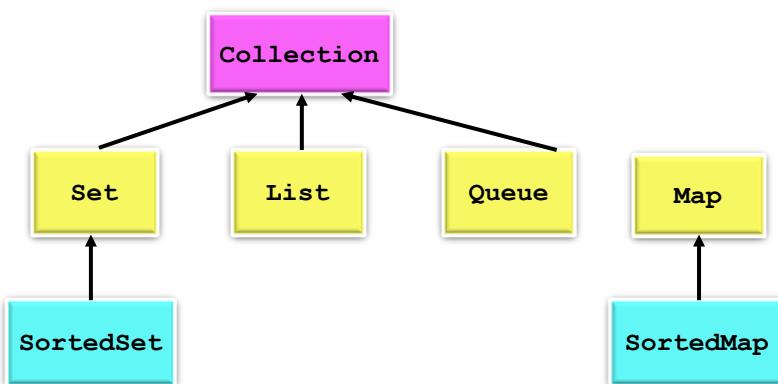
## présentation

- les collections sont un moyen de gérer des données, principalement des objets, au sein d'un même groupe, généralement un objet lui aussi
  - les tableaux constituent la collection la plus simple
  - une collection Java ne peut stocker des variables de type primitif
  - il existe d'autres collections plus élaborées répondant à des besoins différents, comme les listes chaînées, les tables de hachage

## présentation

- dans Java, les collections sont des instances de classes qui, selon les interfaces qu'elles implémentent, déterminent la catégorie à laquelle elles appartiennent
- Java dispose principalement des interfaces **Set**, **List**, **Queue** et **Map** pour décrire les collections, les trois premières héritant de l'interface **Collection**

## interfaces Set, List, Queue et Map



`java.util.Collection` désigne l'interface racine des collections,  
sauf celles de type `java.util.Map`



## l'interface Collection

```
public interface Collection<E> extends Iterable<E>{
 int size();
 boolean isEmpty();
 boolean contains(Object element);
 boolean add(E element);
 boolean remove(Object element);
 Iterator<E> iterator();
 boolean containsAll(Collection<?> c);
 boolean addAll(Collection<? extends E> c);
 boolean removeAll(Collection<?> c);
 boolean retainAll(Collection<?> c);
 void clear();
 Object[] toArray();
 <T> T[] toArray(T[] a);
}
```



## interfaces Set, List, Queue et Map

- le type `java.util.List` représente une collection ordonnée d'éléments pouvant contenir des doublons, et dont les éléments sont généralement accessibles via un indice
- le type `java.util.Set` représente un jeu d'éléments qui ne peut comporter de doublons: un jeu de cartes en est l'exemple type
- le type `java.util.Queue` représente typiquement une collection de type FIFO (First In, First Out)



## interfaces Set, List, Queue et Map

- le type `java.util.Map` représente une table d'éléments, faisant correspondre à chaque élément une clé, laquelle ne peut apparaître en double
- d'autres interfaces, comme `SortedSet` et `SortedMap`, ne représentent qu'un cas particulier des objets `Set` et `Map`, et dans lesquels les éléments sont ordonnés



## classes d'implémentation

| Interfaces         | Tables de hashage                                                         | Arbres équilibrés    | Tableaux                                                            | Listes chaînées                                                                        |
|--------------------|---------------------------------------------------------------------------|----------------------|---------------------------------------------------------------------|----------------------------------------------------------------------------------------|
| <code>Set</code>   | <code>HashSet</code>                                                      | <code>TreeSet</code> |                                                                     | <code>LinkedHashSet</code>                                                             |
| <code>List</code>  |                                                                           |                      | <code>ArrayList</code><br><code>Stack</code><br><code>Vector</code> | <code>LinkedList</code>                                                                |
| <code>Queue</code> |                                                                           |                      |                                                                     | <code>SynchronousQueue</code><br><code>PriorityQueue</code><br><code>DelayQueue</code> |
| <code>Map</code>   | <code>HashMap</code><br><code>Properties</code><br><code>HashTable</code> | <code>TreeMap</code> |                                                                     |                                                                                        |



## collections et généricité

- pour des raisons de sécurité, la généricité a été mise en place dans Java à partir du JDK 1.5
  - elle évite que des objets de types incompatibles ne soient stockés dans une collection, conduisant à des problèmes de transtypage provoquant des exceptions à l'exécution
- les interfaces et les classes liées aux collections sont dorénavant génériques, nécessitant de préciser le type des éléments stockés
  - le compilateur peut ainsi détecter d'éventuelles erreurs



## collections et généricité

### ■ exemple:

```
public class ArrayList<E> extends AbstractList<E> {
 ...
}
```

- E désigne le type des éléments stockés dans la collection
- la création d'une instance d'une classe de collections s'effectue alors en précisant le type des éléments stockés:

```
ArrayList<Personne> l=new ArrayList<Personne>();
```

- Java 7 permet désormais l'utilisation de l'inférence avec <>:

```
ArrayList<Personne> l=new ArrayList<>();
```



## parcours de collections

- le parcours d'une collection peut s'effectuer:
  - au moyen d'une boucle *for-each*
  - à l'aide d'un itérateur
- une boucle *for-each* est le moyen le plus simple de parcourir une collection

```
ArrayList<Personne> l=new ArrayList<Personne>()
.
.
.
for (Personne p : l){
 System.out.println(p);
}
```

- un itérateur est une sorte de curseur que l'on déplace du début à la fin d'une collection afin d'accéder à ses éléments



## parcours de collections

- la méthode **iterator()** d'une collection renvoie un itérateur, instance d'une classe qui implémente l'interface **java.util.Iterator**:

```
public interface Iterator<E> {
 public boolean hasNext();
 public E next();
 public void remove();
}
```

- **next** déplace le curseur et retourne l'élément suivant
- **hasNext** indique s'il existe un autre élément après lui



## parcours de collections

- **exemple**

```
public void print(Collection<Personne> col) {
 Iterator<Personne> it=col.iterator();
 while(it.hasNext()) {
 System.out.println(it.next());
 }
}
```

- si l'on souhaite repartir du début de la collection, il faut demander un nouvel objet **Iterator** par invocation de la méthode **iterator**
- la suppression d'un objet de la collection s'effectue au moyen de la méthode **remove**, après que cet objet ait préalablement été lu par la méthode **next**



## comparaison d'objets

- certaines collections s'appuient sur la comparaison des objets qui y sont stockés:
  - pour détecter les doublons
  - pour trier les objets
- il faut donc équiper la classe des ces objets de méthodes permettant aux collections de les comparer



## comparaison d'objets

- la détection de l'égalité de deux objets s'appuie sur 2 méthodes:
  - **boolean equals (Object obj)**
    - `equals` ne doit retourner true que lorsque l'objet transmis en argument est de la même classe et possède des attributs de mêmes valeurs que ceux de l'objet courant
  - **int hashCode ()**
    - `hashCode` doit retourner un entier (appelé `hashcode`) calculé à partir de la valeur des attributs de l'objet. Lorsque `equals` retourne true, le `hashcode` des objets comparé doit avoir la même valeur



## comparaison d'objets

- les méthodes `equals` et `hashCode` peuvent avantageusement être générées par l'outil de développement (Eclipse, NetBeans)
- exemple:

```
public class Personne{
 private String nom;
 private String prenom;
 private int age;

 public int hashCode() {
 final int prime = 31;
 int result = 1;
 result = prime * result + age;
 result = prime * result + ((nom == null) ? 0 : nom.hashCode());
 result = prime * result + ((prenom == null) ? 0 :
 prenom.hashCode());
 return result;
 }
}
```



## comparaison d'objets

- exemple (suite):

```
public boolean equals(Object obj) {
 if (this == obj) return true;
 if (obj == null) return false;
 if (getClass() != obj.getClass()) return false;
 Personne other = (Personne) obj;
 if (age != other.age) return false;
 if (nom == null) {
 if (other.nom != null) return false;
 } else if (!nom.equals(other.nom)) return false;
 if (prenom == null) {
 if (other.prenom != null) return false;
 } else if (!prenom.equals(other.prenom)) return false;
 return true;
}
```



## comparaison d'objets

- la comparaison d'objets à des fins de tri peut être obtenue par implémentation de l'interface

**java.lang.Comparable:**

```
public interface Comparable<T> {
 public int compareTo(T o);
}
```

- le type T désigne celui de la classe implementée
- les objets que certaines collections doivent pouvoir trier devront voir leur classe implémenter cette interface
  - l'entier retourné par `compareTo` doit être négatif, nul, ou positif selon que l'objet courant est inférieur à, égal à, ou supérieur à l'objet reçu en argument



## comparaison d'objets

- exemple:

- comparaison alphabétique sur le nom:

```
public class Personne implements Comparable<Personne> {
 private String nom;
 private String prenom;
 private int age;

 public int compareTo(Personne o){
 return nom.compareTo(p.nom);
 }

```



## comparaison d'objets

- l'interface `java.util.Comparator` permet de créer des objets de comparaison

```
public interface Comparator<T> {
 public int compareTo(T o1, T o2);
 public boolean equals(Object obj);
}
```

- le tri peut ensuite être effectuée par des méthodes qui exploitent des `Comparator`

- exemple:

```
ArrayList<Personne> liste = new ArrayList<>();
//...
Comparator<Personne> triAlpha = new PersonneTriAlpha();
Collections.sort(liste, triAlpha);
```



## comparaison d'objets

- **exemple (suite):**

```
public class PersonneTriAlpha implements Comparator<Personne> {
 public int compareTo(Personne p1, Personne o2){
 return p1.getNom().compareTo(p2.getNom());
 }
 public boolean equals(Object obj){
 // ...
 }
}

public class PersonneTriAge implements Comparator<Personne> {
 public int compareTo(Personne p1, Personne o2){
 return p1.getAge() - p2.getAge();
 }
 public boolean equals(Object obj){
 // ...
 }
}
```



## collections de type List

- l'interface **List** reprend l'intégralité des méthodes de **Collection**, mais en rajoute quelques autres:

```
public interface List<E> extends Collection<E> {
 public int size();
 public boolean isEmpty();
 public boolean contains(Object elem);
 public boolean add(E elem);
 public boolean remove(Object elem);
 public Iterator<E> iterator();
 public boolean equals(Object o);
 public int hashCode();
 public boolean containsAll(Collection c);
 public boolean addAll(Collection c);
 public boolean removeAll(Collection c);
 public boolean retainAll(Collection c);
 public void clear();
}
```



## collections de type List

- public interface **List** (suite)

```
public Object[] toArray();
public Object[] toArray(Object a[]);
public void add(int index, E elem);
public E remove(int index);
public E get(int index);
public E set(int index, E elem);
public boolean addAll(int index, Collection c);
public int indexOf(Object o);
public lastIndexOf(Object o);
public ListIterator<E> listIterator();
public ListIterator<E> listIterator(int index);
public List<E> subList(int from, int to);
```

- les méthodes supplémentaires qui apparaissent dans cette interface permettent de manipuler les éléments via un indice



## collections de type List

- la méthode **listIterator** permet d'explorer une liste élément par élément au moyen de l'objet **ListIterator** qu'elle retourne.

```
public interface ListIterator<E> extends Iterator<E> {
 public boolean hasNext();
 public E next();
 public boolean hasPrevious();
 public E previous();
 public int nextIndex();
 public int previousIndex();
 public void set(E e);
 public void add(E e);
 public void remove();
}
```

- les objets de type **ListIterator** permettent d'explorer un objet **List** dans les deux directions, tout en autorisant des modifications dans la liste



## collections de type Set

- l'interface **Set** reprend l'intégralité des méthodes de **Collection**:

```
public interface Set<E> extends Collection<E> {
 public int size();
 public boolean isEmpty();
 public boolean contains(Object elem);
 public boolean add(E e);
 public boolean remove(Object elem);
 public Iterator<E> iterator();
 public boolean equals(Object o);
 public int hashCode();
 public boolean containsAll(Collection c);
 public boolean addAll(Collection c);
 public boolean removeAll(Collection c);
 public boolean retainAll(Collection c);
 public void clear();
 public Object[] toArray();
 public Object[] toArray(Object a[]);
}
```



## collections de type Set

```
import java.util.*;
public class TstSet {
 public static void main(String[] args){
 int val;
 TreeSet<Counter> ts=new TreeSet<Counter>();
 for(int i=0;i<20;i++) {
 val=(int) (Counter.MAX*Math.random());
 ts.add(new Counter(val));
 }
 Iterator<Counter> it=ts.iterator();
 while(it.hasNext())
 System.out.println(it.next());
 }
}
```

- un **TreeSet** nécessite que les objets stockés implémentent l'interface Comparable



## collections de type Queue

- les collections de type **Queue** sont adaptées à la communication d'objets entre threads ou applications
  - elles représentent des files d'attente permettant le stockage temporaire d'objets en attente de traitement
- les éléments sont insérés par la queue, et extraits par la tête

```
public interface Queue<E> extends Collection<E> {
 boolean add(E e);
 E element();
 boolean offer(E e);
 E peek();
 E poll();
 E remove();
}
```



## collections de type Queue

- l'interface **Queue** propose des opérations supplémentaires en plus de celles de l'interface **Collection**

- chacune de ces opérations peut être réalisée à l'aide de deux méthodes, l'une propageant une exception en cas d'échec, l'autre retournant une valeur false ou nulle

- Insertion

```
boolean add(E e); // exception en cas d'échec
boolean offer(E e); // false en cas d'échec
```

- Extraction

```
E remove(); // exception en cas d'échec
E poll(); // null en cas d'échec
```

- Inspection

```
E element(); // exception en cas d'échec
E peek(); // null en cas d'échec
```



## collections de type Queue

- **exemple:**

```
import java.util.concurrent.*;
public class TstQueue{
 public static void main(String[] args){
 Personne tpers[]={};
 ConcurrentLinkedQueue<Personne> clq=
 new ConcurrentLinkedQueue<Personne>();
 // remplissage de la file
 for(int i=0;i<tpers.length;i++){
 clq.offer(tpers[i]);
 }
 // extraction de éléments de la collection
 Personne p;
 while((p=clq.poll())!=null)
 System.out.println(p);
 }
}
```



## collections de type Map

- **l'interface Map comporte les méthodes suivantes:**

```
public interface Map<K,V> {
 public int size();
 public boolean isEmpty();
 public V get(Object key);
 public V put(K key, V value);
 public void putAll(Map t);
 public V remove(Object key);
 public boolean containsKey(Object key);
 public boolean containsValue(Object value);
 public void clear();
 public Set keySet();
 public Collection values();
 public Set entrySet();
}
```



## collections de type Map

- interface **Map** (suite)

```
public interface Entry {
 public Object getKey();
 public Object getValue();
 public Object setValue(Object value);
}
```

- les instances des classes qui implémentent l'interface **Map** doivent associer un élément (nommé *value* dans les arguments des méthodes) à un autre objet dit clé (nommé *key*):

```
V put(K key, V value);
```

- la clé doit être unique car elle permet de récupérer l'élément ainsi enregistré dans la structure de données:

```
V get(Object key);
```



## collections de type Map

```
public class Personne {
 private String nom;
 private String prenom;
 private int age;
 private String numss;// numéro de sécurité sociale

 .
 .
 .

 public String getNumss() {
 return numss;
 }

 .
 .
 .
}
```

## collections de type Map

```
public class TstMap{
 public static void main(String[] args){
 Personne tpers[]={ . . . };
 HashMap<String,Personne> hm=
 new HashMap<String,Personne> ();
 for(int i=0;i<tpers.length;i++){
 // la clé est le numéro de SS
 hm.put(tpers[i].getNumss(),tpers[i]);
 }
 System.out.println(hm);
 // recherche d'un élément par son numéro de SS
 Personne p=hm.get("2650165047008");
 if(p != null){
 System.out.println(p);
 }
 }
}
```

## la classe Collections

- la classe `java.util.Collections` contient de nombreuses méthodes utilitaires

```
static <T> boolean
static <T> int
static <T> int
static <T extends Object &
 Comparable<? super T>>
T
static <T> T
static <T extends Object &
 Comparable<? super T>>
T
static <T> T
static <T> boolean
static void
static <T> Comparator<T>
static <T> Comparator<T>
static <T extends Comparable<?
 super T>>
void
static <T> void
```

`addAll(Collection<? super T> c, T... elements)` Adds all of the specified elements to the specified collection.  
`binarySearch(List<? extends Comparable<? super T>> list, T key)` Searches the specified list for the specified object using the binary search algorithm.  
`binarySearch(List<? extends T> list, T key, Comparator<? super T> c)` Searches the specified list for the specified object using the binary search algorithm.  
`max(Collection<? extends T> coll)` Returns the maximum element of the given collection, according to the *natural ordering* of its elements.  
`max(Collection<? extends T> coll, Comparator<? super T> comp)` Returns the maximum element of the given collection, according to the order induced by the specified comparator.  
`min(Collection<? extends T> coll)` Returns the minimum element of the given collection, according to the *natural ordering* of its elements.  
`min(Collection<? extends T> coll, Comparator<? super T> comp)` Returns the minimum element of the given collection, according to the order induced by the specified comparator.  
`replaceAll(List<T> list, T oldVal, T newVal)` Replaces all occurrences of one specified value in a list with another.  
`reverse(List<?> list)` Reverses the order of the elements in the specified list.  
`reverseOrder()` Returns a comparator that imposes the reverse of the *natural ordering* on a collection of objects that implement the Comparable interface.  
`reverseOrder(Comparator<T> cmp)` Returns a comparator that imposes the reverse ordering of the specified comparator.  
`sort(List<T> list)` Sorts the specified list into ascending order, according to the *natural ordering* of its elements.  
`sort(List<T> list, Comparator<? super T> c)` Sorts the specified list according to the order induced by the specified comparator.



## collections et multi-threading

- la plupart des collections ne sont pas "thread-safe" c'est-à-dire qu'elles ne gèrent pas l'accès simultané de la collection par plusieurs threads
- la concurrence d'accès peut alors être gérée:
  - soit au niveau de l'objet qui encapsule la collection
  - soit au niveau de la collection elle-même



## collections et multi-threading

- gestion de la concurrence d'accès au niveau de l'objet qui encapsule la collection
  - les méthodes **synchronized** empêchent l'accès simultané à la collection **personnel** par plusieurs threads simultanément:

```
public class Gestion{
 private HashMap<String, Personne> personnel;
 . . .
 public synchronized getPersonne(String cle){...}
 public synchronized ajoutPersonne(String cle, Personne p){...}
}
```



## collections et multi-threading

- gestion de la concurrence d'accès au niveau de la collection elle-même:

- la classe `java.util.Collections` permet d'obtenir une collection "thread-safe" à partir d'une collection ordinaire

```
public class Collections {
 . . .
 public static <T> Collection<T>
 synchronizedCollection(Collection<T> c);
 public static <T> List<T> synchronizedList(List<T> list) ;
 public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m) ;
 public static <T> Set<T> synchronizedSet(Set<T> s) ;
 . . .
}
```

- exemple:

```
HashMap m = Collections.synchronizedMap(new HashMap(...));
```