JAVA
La maîtrise
2e édition — JAVA 5 et 6

Jérôme Bougeault

TSoft et Groupe Eyrolles, 2003, 2008, ISBN : 978-2-21212250-3

- *Classes et objets*
- *Les constructeurs*
- *Propriétés*
- *Les méthodes*
- *L'héritage*
- *Les interfaces*
- *Les relations*
- *Les packages*
- *Le transtypage*
- *Atelier*

3

Concepts objet avec Java

Objectifs

Java est un langage orienté objet. Il est donc nécessaire, pour bien maîtriser ce langage, de comprendre et d'assimiler les concepts de l'objet.

Ce module présente tous les aspects de l'objet : comment créer et utiliser des objets, quels sont les différents types d'objets, comment réutiliser du code objet, de quelle façon sont construites les applications orientées objet, etc.

Contenu

Nous allons voir ces concepts en partant de leur représentation en UML, et en examinant leur implémentation en Java.

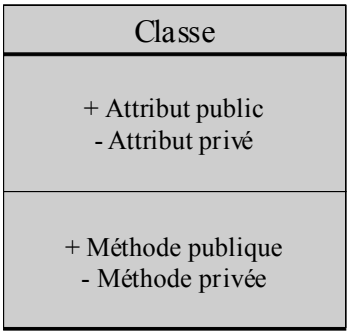
- Créer des objets à partir des classes.
- Modifier les objets.
- Utiliser les services d'un objet.
- L'héritage entre les objets.
- Les classes de concepts abstraits.
- Les relations et les associations entre les objets.
- Les packages de classes.
- Le transtypage des objets.

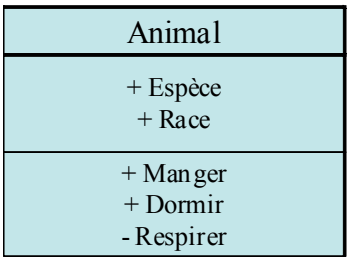
La classe

La classe définit deux types de membres :

- les attributs ;
- les méthodes.

Les membres peuvent être privés ou publics.





Dans cet exemple, la classe Animal possède :

- des attributs : Espèce et Race ;
- des méthodes : Manger, Dormir et Respirer.

La méthode Respirer est privée (seule cette classe peut l'appeler).

Lorsque l'on écrit des programmes, on comprend bien la différence entre le source du programme et son instance d'exécution :

- On commence par écrire un programme (dans un langage de programmation quelconque), qui sera composé d'instructions qui sont des opérations sur des variables. Ce programme est stocké dans un fichier de caractères (les langages informatiques sont généralement proches du langage naturel que nous parlons).
- Lorsque le programme est écrit, il peut alors être exécuté. Les variables prennent leur place dans la mémoire, et les instructions sont transmises au microprocesseur.

Un programme informatique est donc construit à partir de deux choses : des variables en mémoire et des instructions.

En programmation objet, c'est la même chose. La différence réside simplement dans le fait que l'on peut (et que l'on doit) bâtir les applications à partir de briques que l'on appelle les objets.

Les objets sont de petits programmes qui possèdent leurs propres variables et leurs propres instructions.

Ainsi donc, au lieu de développer un seul gros programme, nous allons en développer de nombreux petits, qui communiqueront ensemble.

Le processus de réalisation des objets est le même que celui des programmes classiques :

- On écrit le code de l'objet. Il contient des déclarations de variables et des instructions. On appelle cela la **classe**. Une classe est donc simplement un programme source, et il tient dans un fichier.
- Puis on exécute le programme. La classe s'exécute sur l'ordinateur, son instanciation s'appelle alors un **objet**. Un objet est donc le programme tel qu'il s'exécute dans la mémoire de l'ordinateur.

Les membres

Un objet contient des variables et du code, déclarés et définis dans la classe. Ces deux composants s'appellent les membres. On a donc deux types de membres :

- Les **Attributs** : aussi appelés **Champs**, **Propriétés** ou **variables d'instance**, ce sont les variables définies (déclarées) dans la classe.
- Les **Méthodes** : les instructions du code de la classe sont regroupées dans des fonctions. On appelle ces dernières les méthodes de l'objet. On dit qu'elles définissent le **comportement de l'objet**.

En Java, les classes sont définies dans des groupes d'instructions (entre accolades) précédés de la déclaration `class` suivie du nom de la classe :

```
class NomDeLaClasse {
  // Définition des propriétés (variables) et des
  // méthodes (code) de la classe
}
```

Le nom de la classe doit être unique. C'est ce nom qui sera utilisé pour créer des objets à partir de la classe.

La notation UML

Les classes sont définies en phase de conception. On s'appuie sur cette notation pour concevoir les applications.

Comme nous le verrons, un programme objet est composé d'un grand nombre de classes (et donc d'objets) qui interagissent entre eux. De plus, les relations qui existent entre ces objets sont nombreuses et mettent en œuvre de nombreux concepts. La complexité d'une application objet peut être telle qu'une description par le langage peut s'avérer très difficile.

Comme on dit souvent, un petit schéma vaut mieux qu'une longue explication. C'est en partant de ce principe qu'a été imaginée la notation UML.

Cette notation permet de représenter tous les concepts de l'objet au travers d'une symbolique graphique. Nous verrons cette symbolique dans les transparents de ce module.

Le transparent ci-avant montre la représentation d'une classe en notation UML. Le symbole est donc un rectangle séparé en trois parties :

- En haut : le nom de la classe.
- Au dessous : la liste des propriétés.
- Encore au dessous : la liste des méthodes.

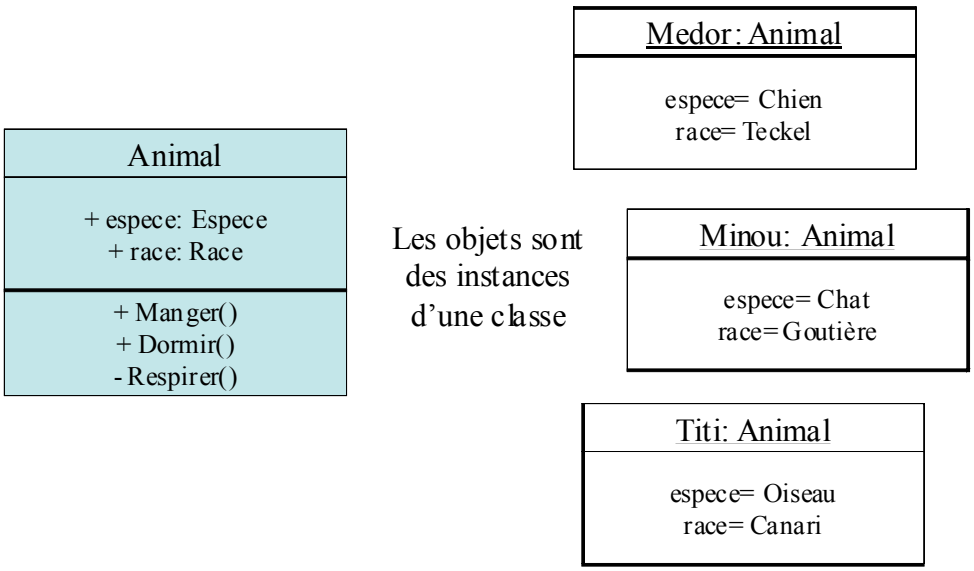
L'exemple animal qui est présenté possède deux propriétés et trois méthodes :

`Espèce` et `Race` est propre à chaque animal, et ces champs sont publics car on veut pouvoir accéder à ces informations de l'extérieur de l'objet.

`Manger`, `Dormir` et `Respirer` sont des méthodes, le code est donc le même pour tous les objets `Animal`. Les animaux mangent, dorment et respirent donc tous, et tous de la même manière.

On note que `Respirer` est privé, cette méthode est donc interne à chaque animal. En effet, la respiration est un mécanisme qui ne peut être provoqué de façon externe.

Les objets



Les objets sont créés à partir des classes, la classe est donc une sorte de moule. L'objet est donc moulé à partir de la classe.

On peut aussi définir la classe comme un ensemble d'éléments (appelés objets) partageant les mêmes comportements et une même structure.

Un objet est appelé **instance de classe**. Il est créé (on dit aussi construit) à partir d'une classe, et réside dans la mémoire centrale de l'ordinateur.

Un objet est un module logiciel encapsulé. Il possède :

- Un **état** : ce sont les valeurs de ses attributs qui sont définies dans la classe.
- Un **comportement** : c'est l'ensemble de ses méthodes, définies et implémentées dans la classe.
- Une **identité** : c'est ce qui permet de l'identifier pour le manipuler.

Tous les objets issus d'une même classe ont strictement la même structure et les mêmes comportements.

Les constructeurs

En Java, c'est l'opérateur `new` qui permet de créer des objets. Il prend en argument le type de l'objet (c'est à dire le nom de la classe) et éventuellement des paramètres qui pourront être utilisés par la méthode de création de l'objet, ce que l'on appelle le **`constructeur`**.

Exemple :

```
Animal ami= new Animal();
```

Cette ligne effectue les opérations suivantes :

- Déclaration d'une variable a contenant la référence d'un objet de type `Animal`.
- Assignation de `a` : on met la référence de l'objet créé (opérateur `new`).

Si nous reprenons l'exemple de transparent, l'implémentation de la classe `Animal` sera la suivante :

```
public class Animal {
  // Définition des propriétés
  public String espece; // L'espèce est définie par une chaîne
  public String race;   // La race aussi
  // Définition des méthodes
  public void manger() {
    // Instructions pour manger
  }
  public void dormir() {
    // Instructions pour dormir
  }
  private void respirer() {
    // Inspirer, expirer
  }
}
```

Accès aux membres

Pour accéder aux variables d'un objet, ou invoquer (exécuter) une de ses méthodes, on a besoin de sa référence (obtenue par le `new`).

A partir de sa référence, on va "pointer" le membre à l'aide de l'opérateur point (`.`).

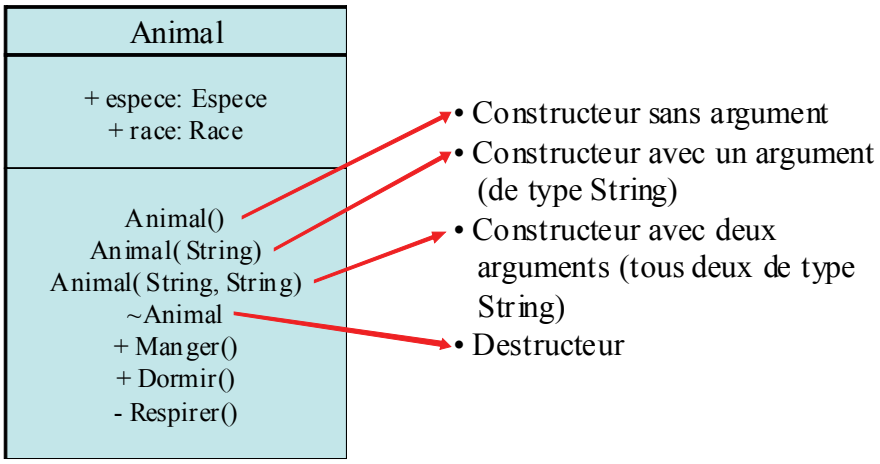
Exemple :

```
ami.race= "Caniche"; // Initialisation
System.out.println( "Race: "+ami.race); // Récupération
ami.dormir(); // Invocation de la méthode
```

La création et l'initialisation des trois objets `Médor`, `Minoux` et `Titi` se feront de la manière suivante (dans le `main` ou dans n'importe quelle méthode d'un autre objet) :

```
// Déclaration de trois variables objet:
Animal medor;
Animal minou;
Animal titi;
// On crée les trois nouveaux animaux:
medor= new Animal();
medor= new Animal();
titi= new Animal();
// On spécifie leurs propriétés:
medor.espece= "Chien"; medor.race= "Teckel";
minou.espece= "Chat"; minou.race= "Gouttière";
titi.espece= "Oiseau"; titi.race= " Canari";
```

Les constructeurs



L'opérateur `new` permet de créer des objets.

Le processus de création dans la machine virtuelle Java s'effectue en quatre étapes :

- Réservation de la mémoire : l'objet à créer est typé, la JVM connaît donc la taille mémoire à réserver pour y mettre l'objet.
- Création d'un objet "vierge" dans l'espace mémoire alloué (les propriétés sont initialisées à 0).

Initialisation de l'objet : appel d'un `constructeur` défini dans l'objet.

Renvoi de la référence de l'objet (son adresse) à l'argument de gauche de l'opérateur `new`, qui doit être une variable du même type que l'objet créé.

Le constructeur est une sorte de méthode, qui sera appelée uniquement au moment de la création de l'objet.

Il permet d'effectuer des opérations d'initialisation de l'objet.

Il est défini dans la classe.

On peut avoir autant de constructeurs que l'on souhaite dans une classe. Toutefois, deux constructeurs d'une même classe ne peuvent avoir le même nombre et les mêmes types d'arguments.

Exemple :

```
class Animal {
  // Définition des propriétés
  public String espece; // L'espèce est définie par une chaîne
  public String race;   // La race aussi
  // Constructeurs
  public Animal( String e, String r) {
    espece= e; // Initialisation de la propriété espece
```

```
    race= r;   // Initialisation de la propriété race
  }
  // Constructeur sans argument
  public Animal() {
    espece=""; race="";
  }
}
```

En Java, les constructeurs doivent être définis en respectant impérativement les deux règles suivantes :

- Ils portent obligatoirement le même nom que celui de la classe (attention aux majuscules et aux minuscules, Java est “case sensitif”).
- Ils sont définis comme des méthodes, toutefois ils ne sont pas typés (ils ne retournent pas de valeur).

Lors de la création d'un objet, seul un constructeur est appelé, celui qui a été utilisé lors de sa construction.

Exemple :

```
Animal medor;
// Utilisation du constructeur qui prend deux arguments
medor= new Animal( "Chien", "Teckel");
// Utilisation du constructeur sans argument:
minou= new Animal();
```

this et null

`this` est un mot clé qui représente l'objet dans lequel on est.

Il a deux utilisations :

- Utilisé comme référence, c'est la référence de l’objet dans lequel on est.
- Utilisé comme méthode, c'est un des constructeurs de l'objet dans lequel on est. Ce constructeur ne peut être appelé que par un autre constructeur de ce même objet.

Exemple :

```
class Animal {
  // Définition des propriétés
  public String espece; // L'espèce est définie par une chaîne
  public String race;   // La race aussi
  // Constructeurs
  public Animal( String espece, String race) {
    // Utilisation de this comme référence sur moi-même
    // Permet de faire le distingo entre espece variable locale
    // et espece variable d'instance (this.espece)
    this.espece= espece;
    this.race= race;
  }
  // Constructeur sans argument
```

```
  public Animal() {
    // Appel du constructeur par défaut
    this( "", "");
    // Ci-dessous est tentant mais syntaxiquement incorrect
    // Animal( "", "");
  }
}
```

`null` est la valeur d'une variable objet non initialisée.

A la création des objets, toutes leurs propriétés sont initialisées à 0. Cela signifie que :

- Les propriétés ayant un type primitif (`int`, `float`, `boolean`…) sont initialisées à 0 (pour les entiers), 0.0 (pour les nombres à virgule flottante) ou encore `false` (pour les booléens).
- Les propriétés ayant un type objet sont initialisées à `null`.

Exemple :

```
String s; // s est égal à null
String s= new String( "Hello"); // s n'est pas à null
```

Constructeur par défaut

La première classe `Animal` que nous avons fait en début de module ne possédait pas de constructeur.

Si on ne définit pas de constructeur dans une classe, alors Java nous en met un, par défaut, et de façon transparente. Ce constructeur par défaut ne possède pas d'argument.

Donc, une classe possède toujours au moins un constructeur.

Remarque :
Java ne met un constructeur par défaut que dans les classes qui n'en ont pas. Ainsi, si on définit un constructeur qui prend des arguments, cette classe n'aura alors plus de constructeur sans argument.

Le destructeur

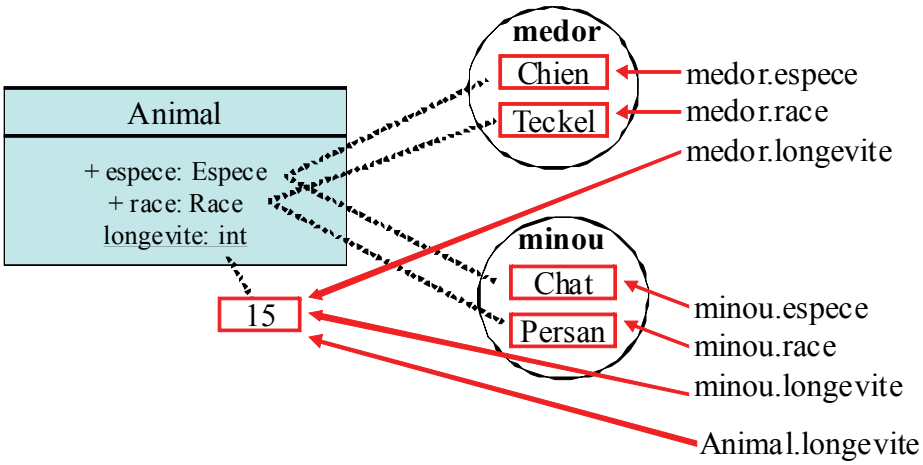
Contrairement à certains autres langages, Java ne possède pas de destructeur. En effet, la destruction des objets est à la charge du Garbage Collector.

Toutefois, il existe une méthode qui est appelée par le Garbage Collector lorsque celui-ci va détruire l'objet.

```
public void finalize();
```

Cette méthode peut être implémentée si on souhaite y mettre du code.

Zoom sur les propriétés



Les variables d'instance sont stockées dans les objets.
Les variables de classe sont stockées dans les classes.

Ce sont les variables internes à chaque objet créé à partir de la classe. On appelle cela aussi des **champs**, des **attributs** ou encore des **variables d'instance**.

Ces variables peuvent être de type simple, mais aussi de type objet. Elles contiennent alors des références d'autres objets.

Les modifier est aussi appelé « configurer l'objet ».

Elles sont définies dans la classe, mais initialisées dans l'objet. Chaque objet a donc des valeurs différentes dans ses champs.

Comme on le voit sur la figure, elles sont stockées dans la mémoire de chaque objet.

Attributs de classe

Une variable de classe est globale à toutes les instances de la classe. Sa valeur est donc partagée parmi tous les objets construits avec cette classe.

Sa valeur n'est pas stockée dans l'espace mémoire des objets, mais dans la classe elle même. On peut donc accéder à sa valeur en dehors de toute instance de la classe.

Sa représentation UML est symbolisée simplement par une propriété soulignée (voir le transparent).

En Java, les attributs **static** sont déclarés à l'aide du modificateur `static`.

Exemple :

```
class Animal {
  Espece espece; // Java est case-sensitif, d'où la
  Race race;     // distinction entre Espece et espece et
                 // entre Race et race
  static int longevite;
}
```

On peut accéder à une propriété `static` à partir d'une instance, mais aussi directement à partir de la classe :

```
Animal medor= new Animal();
medor.longevite= 15;
Animal minou= new Minou();
// minou.longevite = medor.longevite = 15
Animal.longevite= 20;
// Maintenant minou.longevite = medor.longevite = 20
```

Voici un célèbre exemple intéressant à décortiquer :

```
System.out.println( "Hello");
```

On voit dans cette ligne :

- `println` est une méthode prenant en argument une chaîne de caractères.
- `out` est un objet dans lequel est invoquée la méthode `println`.
- `System` est une classe dans laquelle out est un attribut `static`.

Remarque :
Afin de faciliter la lecture du code, et notamment de distinguer les noms des classes des noms des objets, Sun conseille de suivre la convention qui consiste à toujours donner aux classes un nom qui commence par une majuscule, et aux objets, attributs et méthodes un nom qui commence par une minuscule.
Je vous conseille vivement de suivre cette convention.

En tenant compte de cette remarque, on en déduit que `System` est bien une classe, donc `out` est obligatoirement une propriété statique.

Un autre exemple est la méthode `main`.

Le démarrage d'un programme Java passe par le démarrage de la machine virtuelle Java, en passant en argument le nom d'une classe. Cette classe doit impérativement avoir un point d'entrée sous la forme d'une méthode statique appelée `main` (ce qui veut dire principale en anglais).

La signature de cette méthode doit être précisément la suivante :

```
public static void main( String [] args);
```

Cette méthode prend en argument un tableau de chaînes de caractères qui sont les arguments passés en paramètres sur la ligne de commande.

Attributs et méthodes `final`

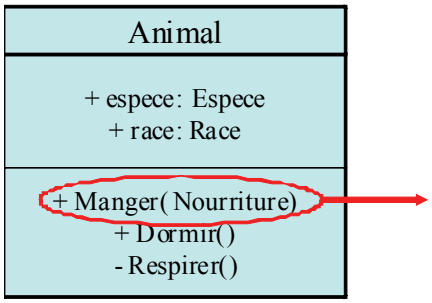
Le modificateur `final` permet de spécifier un membre qui ne pourra pas être modifié.

C'est tout simplement une constante. Il sera donc nécessaire de toujours l'initialiser lors de sa déclaration.

Exemple :

```
public final double PI=3.141592;
```

Zoom sur les méthodes



La signature d'une méthode :
- son nom
- le nombre d'arguments
- les types d'arguments

Les méthodes accèdent à toutes les propriétés de l'objet.
Les méthodes de classe n'accèdent qu'aux variables de classe.

Ce sont les actions élémentaires que peut effectuer un objet. On appelle cela aussi les **services de l'objet**.

L'ensemble des méthodes définissent le **comportement des objets**.

Pratiquement, ce sont des fonctions écrites à l'aide du langage de programmation.

Les méthodes permettent aux objets de communiquer entre eux. On dit souvent que les objets communiquent par messages. Le mécanisme repose sur l'appel des méthodes, les messages sont alors les arguments.

On utilise le point comme opérateur d'accès aux méthodes, comme pour les propriétés.

Exemple :

```
Animal medor= new Animal("Chien", "Teckel");
o.toString();          // Invocation de la méthode toString de o
int i= o.sonEntier;    // Accès à l'attribut sonEntier de o
```

La signature de la méthode

Elle représente son nom, mais aussi le nombre et les types de ses arguments. Dans une classe, deux méthodes ne peuvent pas avoir la même signature, ce qui veut dire que deux méthodes peuvent avoir le même nom, à condition d'avoir des arguments différents.

Exemples :

```
public class Animal {
  // Propriétés
  public String espece, race, longevite;
  // Méthodes
  public boolean estDeRaceConnue() {
```

```
    if( race != null)
      return true;
    else
      return false;
  }
  public void afficher() {
    System.out.println( "Espece: "+espece);
  }
  public void afficher( String nom) {
    System.out.println( "L'animal "+nom+" est un "+espece);
  }
  // Ci-dessous erreur car signature déjà utilisée
  public boolean afficher() {
    return true;
  }
}
```

Dans l'exemple ci-dessus, on a défini trois méthodes `afficher`.

Bien qu'elles possèdent le même nom, les deux premières n'ont pas les mêmes arguments (la première n'en a pas et la deuxième en a un de type `String`). Elles n'ont donc pas la même signature, et peuvent cohabiter dans la même classe.

Par contre, la troisième, bien qu'elle ne soit pas du même type que les deux premières (`boolean` au lieu de `void`), pose un problème car sa signature est la même que la première (pas d'argument). Ce problème sera détecté à la compilation qui ne pourra pas s'effectuer.

Le polymorphisme

Pourquoi faire plusieurs méthodes qui possèdent le même nom ?

C'est la base d'une notion que l'on appelle le polymorphisme. Ce terme vient de deux mots grecs : Poly et Morphe. On peut les traduire par "différentes formes".

Le principe est d'implémenter de différentes manières des services, destinés à faire la même chose, parce qu'ils s'appuient sur des données différentes.

Dans notre exemple, les méthodes `afficher` sont destinées à faire la même chose : afficher. Toutefois, les arguments étant différents, le code le sera aussi.

Passage des arguments

En Java, les arguments sont passés par valeur.

Cela signifie que lorsque l'on appelle une méthode en lui passant des variables en argument, c'est la valeur de la variable et non pas son adresse qui est passée. Ainsi, si la méthode modifie l'argument, la valeur initiale de la variable ne sera pas touchée.

Exemple :

```
public void ajouter( int a, int b) {
  a=a+b;
}
int e= 3; // e = 3
```

```
ajouter( e, 2);
// e = 3 et non pas 5
```

Toutefois, il y a un effet de bord intéressant à signaler : les variables objets contiennent les références (adresses) de ces objets. Passer en argument une variable objet, c'est passer son adresse (et non pas une copie).

Par exemple, supposons la classe suivante :

```
public class Entier {
  int valeur;
}
```

Examinons le code ci-dessous :

```
public void ajouter( Entier e, int montant) {
  e.valeur = e.valeur+montant;
}
Entier e;
e.valeur= 3;
ajouter( e, 2);
// e.valeur est maintenant à 5!
```

On voit bien que l'objet passé en argument a été modifié.

La variable `e` passée en argument n'a pas été modifiée (elle possède toujours la même référence de l'objet qu'elle pointe). Par contre, c'est le contenu de l'objet qui a été modifié par la méthode `ajouter`.

Les variables locales

On peut définir des variables de travail à l'intérieur d'une méthode. Ces variables seront soumises aux règles suivantes :

- Elles ne sont visibles qu'à l'intérieur de la méthode.
- Leur contenu est perdu dès que l'on quitte la méthode (elles ne sont pas conservées entre deux appels à la même méthode).
- Elles ne sont pas initialisées à `0` par défaut.

Exemple :

```
public class TestVarLocales {
  static int b= 0;
  public static void fonction(){
    int a= 0; // Initialisation obligatoire variables locales
    System.out.println( "a est à: "+a+" et b à: "b);
    if( a == 0) a= 1;
    if( b == 0) b= 2;
  }
  public static void main( String [] args) {
    fonction();
    fonction();
  }
}
```

Ce programme donne :

```
a est à: 0 et b à: 0
a est à: 0 et b à: 2
```

On voit donc bien que la valeur de la variable locale `a` n'est pas conservée entre deux appels à la fonction, alors que celle de la variable d'instance `b` est bien conservée.

On remarque que la fonction et la variable `b` sont déclarés en `static`, car elles sont appelées par la méthode `main` qui est `static` aussi (nous verrons cela plus loin).

Enfin, il faut noter l'obligation d'initialiser les variables locales avant de les utiliser. Java n'initialise pas automatiquement les variables locales à `0` comme c'est le cas pour les variables d'instance. Le compilateur effectue d'ailleurs une vérification dans le code. Si on déclare une variable locale non initialisée, dès la première instruction qui cherche à récupérer son contenu, une erreur sera générée par le compilateur.

Type et retour des méthodes

Les méthodes sont typées. Elles peuvent avoir un type parmi :

- Tous les types primitifs (`int`, `float`, `boolean`, etc.).
- Tous les types objet (`String`, `Date`, `Animal`, `LecteurDeMonLivre`, etc.).
- `void`, c'est à dire qu'elles ne retournent pas de valeur.

A l'exception des méthodes de type `void`, les méthodes doivent obligatoirement retourner une valeur de leur type.

On utilise pour cela l'instruction `return` suivie de la valeur à retourner (qui peut être dans une variable).

Exemples :

```
public int somme( int a, int b) {
  return a+b;
}
public String ditBonjourALaDame() {
  return "Bonjour Madame";
}
public Animal rendUnChien( String race) {
  Animal a= new Animal( "Chien", race);
  return a;
}
```

Remarque :
Pour les "procédures" (méthodes de type `void`, on peut aussi utiliser l'instruction `return` (sans valeur de retour) simplement pour quitter la méthode et revenir à l'appelant.

Accesseurs et mutateurs

Ce sont des méthodes permettant l'accès en lecture (accesseur) et en écriture (mutateur) aux attributs.

Il peut être intéressant de déclarer tous les attributs en privé, et de permettre l'accès à certains d'entre eux au travers de méthodes publiques.

Cela permet par exemple de :

- Comptabiliser leur accès.
- Définir une logique d'accès implémentée dans la méthode.
- Tracer dans une log les accès.
- Vérifier les droits d'accès par un mécanisme de sécurité implémenté dans la méthode.

Généralement, ces méthodes portent des noms qui sont composés de `get` (en lecture) ou de `set` (en écriture) suivi du nom de la propriété.

Exemples :

```
public class Animal {
  // Propriétés en private
  private String espece; private String race;
  // Méthodes d'accès à l'espece
  public String getEspece() {
    return espece();
  }
  public void setEspece( String espece) {
    this.espece= espece;
  }
  // Accès à la race en lecture seule (pas de set…)
  public String getRace() {
    return race;
  }
}
```

Remarque :
Une méthode `public` peut accéder à un membre `private`, et vice-versa.

Les méthodes à nombre variable d'arguments

Cette fonctionnalité qui existe en C est maintenant supportée par Java. Elle permet de spécifier un nombre indéfini d'arguments d'un même type.

Dans cet exemple, notez la forme particulière de la boucle for pour récupérer l'ensemble des arguments disponibles :

```
public static int somme( int ... args) {
  int somme=0 ;
  for( int x : args) {
    somme+=x;
  }
  return somme;
}
```

Cet exemple pourra être invoqué de la manière suivante :

```
System.out.print( "somme de 3, 4, 5,12 et 7= "
    +somme(3, 4, 5, 12, 7));
```

La variable contenant les arguments sera en fait un tableau. L'exemple ci-dessus est équivalent à ceci :

```
public static int somme( int … args) {
  int somme=0 ;
  for( int n=0 ; n < args.length ; n++)  {
    somme+= args[n];
  }
  return somme;
}
```

Méthodes de classe

Elles sont spécifiées par le modificateur `static` (comme pour les variables de classes).

Ces méthodes peuvent, comme pour les variables de classe, être utilisées au travers d'une instance de la classe, ou directement dans la classe.

Cela explique qu'elles ne peuvent faire référence qu'à des variables locales ou à des variables de classe, et qu'elles ne peuvent invoquer que d'autres méthodes de classe.

Exemple :

```
import java.util.Date;
public class Random{
  private static int random= (int)(new Date().getTime());
  public static int random() {
    random= random ^ (random * random);
    return random;
  }
  public static void main( String [] args) {
    for( ;;)
      System.out.println( "Aléatoire: "+Random.random());
  }
}
```

Cet exemple est un petit générateur de nombres aléatoires.

La méthode de calcul est `static`. Elle peut donc être appelée directement à partir de la classe, c'est d'ailleurs ce qui est fait dans le main :

```
System.out.println( "Aléatoire: "+Random.random());
```

Cette ligne se trouve dans un `for( ;; )` qui est donc une boucle sans fin (il sera nécessaire de faire un CTRL+C pour stopper le défilement de nombres aléatoires).

La méthode `random()` utilise la variable `static random` (on peut donner à une propriété le même nom qu'une méthode). Cette variable, de type `int`, est initialisée à la déclaration. Cette initialisation est effectuée automatiquement dès la mie en mémoire de la classe (au démarrage de la JVM).

La valeur d'initialisation de `random` est le résultat de l'invocation de la méthode `getTime()` de l'objet créé à partir de la classe `java.util.Date`. Nous parlerons des packages et de la classe `Date` un peu plus tard, considérons simplement que la

variable `random` est initialisée au nombre de millisecondes depuis le 1er janvier 1970 au moment où le programme a été démarré.

Chaque appel à la méthode `random` va modifier la variable `random` en effectuant un ou exclusif entre sa valeur et le résultat de son carré. Cela nous donnera à chaque fois une valeur différente qui peut s'apparenter à un nombre aléatoire (même si ce n'est pas tout à fait aléatoire !

Enfin, on remarque le `cast` à l'appel de `getTime()` :

```
random= (int)(new Date().getTime());
```

En effet, `getTime()` est une méthode qui retourne un `long`. Il y aura perte d'information (cela ne nous gène pas car on cherche simplement un nombre aléatoire).

Pour résumer et bien comprendre quand les variables sont initialisées :

- Les variables d'instance déclarées avec une initialisation sont initialisées à la création de l'instance (création de l'objet avec l'opérateur `new`).
- Les variables de classe déclarées avec une initialisation sont initialisées au démarrage de la machine virtuelle (démarrage de l'application).

Les blocs d'initialisation statique

L'initialisation d'une variable peut nécessiter un code un peu plus complexe qu'une simple assignation.

Dans le cas de variables d'instance, les constructeurs sont faits pour cela.

Dans le cas de variables de classe, on dispose des blocs d'initialisation statique.

Ils permettent d'exécuter du code au moment du démarrage de la machine virtuelle Java.

Ils sont simplement positionnés dans la classe (entre deux accolades). S'il en existe plusieurs, ils seront alors exécutés en séquence.

On peut faire le parallèle entre ces blocs et les constructeurs, aux différences que :

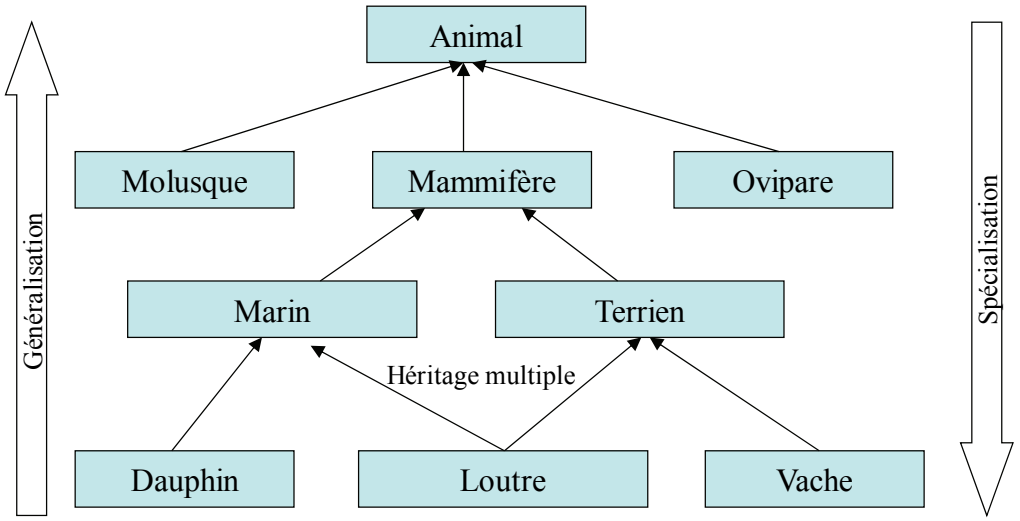
- Ils ne reçoivent pas de paramètres (mais on peut toutefois aller les chercher dans un fichier de configuration : nous verrons notamment plus tard la classe `Property`).
- Ils ne sont exécutés qu'une fois au démarrage de la JVM (alors que le constructeur est exécuté à chaque création d'objet).

Exemple :

```
public class DebugLog {
  static OutputStream sortie; // Flux d'écriture
  static {  // Début d'un bloc d'initialisation
    sortie= System.out;
  }
}
```

Dans cet exemple, on initialise la propriété sortie à l'objet `OutputStream` qui spécifie la sortie standard (les entrées/sorties sont vues dans un prochain module).

L'héritage



L'héritage permet de créer des classes en réutilisant des classes déjà existantes. Les nouvelles classes ainsi créées peuvent contenir de nouveaux membres (méthodes et propriétés) ou redéfinir des propriétés ou des méthodes déjà définies dans les classes dont on hérite.

Cela permet donc d'étendre les possibilités d'une classe. D'où l'emploi du terme : étendre une classe.

Remarque :

- Une classe ne peut hériter d'elle même, évidemment.
- Une classe ne peut hériter d'une classe qui elle même hérite déjà d'elle (car là on se mord la queue).

La spécialisation

Cela consiste à partir de classes très générales, et d'utiliser l'héritage pour créer des classes de plus en plus particulières.

Par exemple, partir de la classe `Humain`, l'étendre vers la classe `Travailleur`, puis `Salarié`, puis `Fonctionnaire`, etc.

La généralisation

C'est le chemin inverse de la spécialisation : on part d'un grand nombre de classes très spécialisées, et on essaie d'en extraire les concepts communs afin de construire des classes très générales dont hériteront les classes plus spécialisées.

Cette démarche peut faciliter la création de classes très réutilisables.

La recherche d'un membre dans la hiérarchie

Lorsque l'on cherche à atteindre un membre d'un objet (attribut ou méthode), la machine virtuelle le recherche d'abord dans l'objet, puis s'il n'est pas trouvé, dans le père de l'objet, et ainsi de suite jusqu'à la classe la plus haute dans la hiérarchie.

Redéfinir un membre, c'est donc l'occulter derrière le nouveau membre défini.

Les méthodes et les classes "final"

Nous avons déjà vu le modificateur `final`.

Pour une méthode, cela signifie qu'elle ne pourra pas être redéfinie dans les héritiers de la classe. Son fonctionnement sera donc figé, elle est finale.

Pour une classe, c'est la classe elle même qui ne pourra pas être étendue, elle ne pourra avoir d'héritier.

L'héritage multiple

L'héritage multiple permet d'hériter de plusieurs classes. C'est assez dangereux car cela peut générer des conflits d'implémentations.

Dans l'exemple de notre moteur, supposons que la méthode `demarrer()` soit implémentée à la fois dans `Véhicule à essence` et dans `Véhicule électrique`. Si j'invoque cette méthode dans `Voiture hybride`, et que cette méthode n'y est pas redéfinie, alors laquelle sera invoquée ?

En Java, il a été décidé de ne pas permettre l'héritage multiple.

Implémentation de l'héritage en Java

En Java on utilise le mot clé `extends` qui permet de spécifier la classe dont on hérite.

Exemple :

```
public class Animal {
  public void seDeplacer() {
  }
}
```

```
public class Mammifere  extends Animal {
}
```

Voici la classe `Mammifere` qui hérite de `Animal`. On remarque qu'aucune implémentation n'a été mise dans le corps de la définition de la classe `Mammifere`.

Si j'invoque la méthode `seDeplacer` dans un objet créé à partir de la classe `Mammifere`, alors c'est le code de la méthode implémentée dans `Animal` qui sera exécuté.

```
public class Marin extends Mammifere {
  public void seDeplacer() {
    // Plouf, floc floc je nage
  }
}
```

Dans ce cas, `Marin` hérite de `Mammifere`, mais **redéfinit** la méthode `seDeplacer()`.

Le polymorphisme

L'héritage permet de favoriser le polymorphisme dynamique.

Il existe deux formes de polymorphisme :

- Le polymorphisme statique : plusieurs méthodes d'une même classe qui ont une même fonction possèdent un même nom, mais ont des arguments différents, donc une implémentation différente (nous l'avons déjà vu précédemment).
- Le polymorphisme dynamique : une méthode ayant une même signature (même nom et mêmes types d'arguments) sera implémentée dans différentes classes, donc de différentes manières suivant la logique de chaque classe.

Avec l'héritage, on peut définir des méthodes qui seront redéfinies dans toutes les classes héritières pour s'adapter à la logique des classes.

Par exemple, la méthode `seDeplacer()` définie dans `Animal` possédera évidemment une implémentation différente suivant s'il s'agit d'un `Molusque`, d'un `Mammifere Marin` ou encore d'un `Mammifere Terrien`.

Types des objets

La classe d'un objet représente aussi son type. Un objet créé à partir de la classe `Marin` est de type `Marin`.

Avec l'héritage, un objet est aussi du type des classes dont il hérite.

Dans notre exemple, un objet `Marin` est aussi de type `Mammifere` ainsi que du type `Animal`.

On peut d'ailleurs dire dans notre langage naturel : "Un mammifère est un animal".

La classe Object

Lorsqu'une classe n'hérite de rien, elle hérite en fait, par défaut, de la classe `Object`.

Cette classe définie dans Java, est la classe racine de Java. Toutes les classes héritent directement ou indirectement de la classe `Object`.

Dans l'exemple ci-dessus, `Animal` hérite de `Object`.

En Java, il existe un certain nombre de méthodes polymorphes que l'on trouve dans toutes les classes car elles sont implémentées dans `Object`. Citons par exemple :

- `public String toString();` retourne une chaîne de caractères, représentation textuelle de l'objet (le contenu dépendra donc directement de la nature de l'objet).
- `public boolean equals( Object);` permet de comparer l'objet à un autre objet. Cette implémentation dépend là aussi directement de la logique de l'objet.
- `public int hashCode();` rend un code de hashage de l'objet.
- `protected Object clone();` rend un objet copie conforme de l'objet, etc.

Toutes les méthodes ci-dessus sont implémentées dans la classe `Object`, mais peuvent être redéfinies dans les classes que l'on crée, afin de correspondre à leurs logiques.

Super

Le mot clé `super` définit l'instance de l'objet de la classe dont on hérite directement. Super, c'est le père!

On peut l'utiliser pour accéder aux membres définis dans la classe dont on hérite, on peut aussi l'utiliser pour appeler un constructeur du père.

Exemple :

```
public class Heritier extends Ancetre {
  int age;
  public Heritier( String nom) {
    super( nom);  // O.K. si ce constructeur existe dans
                  // la classe pere
  }
```

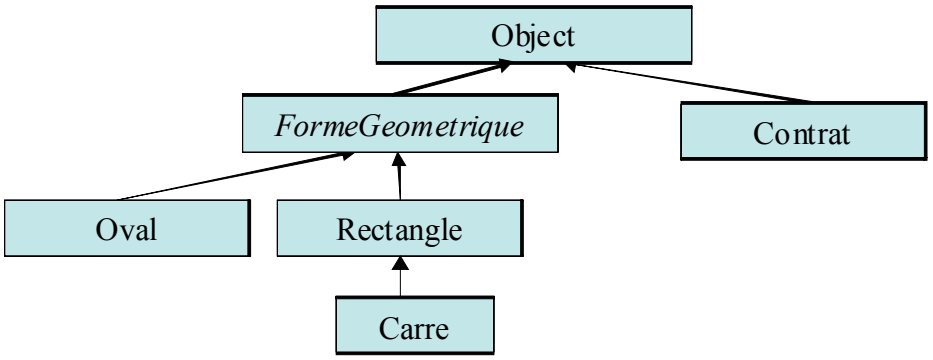
Remarque :
L'utilisation de super pour invoquer un constructeur du père doit se faire sous certaines règles et contraintes très importantes :

- On ne peut pas appeler le constructeur du père en dehors d'un constructeur.
- On ne peut appeler qu'un seul constructeur du père, et obligatoirement en première instruction du constructeur du fils (afin de respecter la règle d'ordre d'exécution des constructeurs : du plus ancien vers le plus jeune. Honneur aux anciens!).
- Par défaut, s'il n'y a pas d'appel à un constructeur du père, tout constructeur appellera obligatoirement le constructeur par défaut du père, celui qui n'a pas d'argument : `super();`.

```
  public Heritier() {
    super(); // Ligne inutile (lire le point 3 de la remarque
             // ci-dessous
  }
  public Heritier( int n) {
    age= n;
    super(); // Erreur! Lire le point 2 de la remarque
             // ci-dessous. Il faut inverser ces 2 lignes
  }
  public int getAgeDuPere() {
    super();  // Erreur! Lire le point 1 de la remarque
    return super.age(); // O.K. si l'attribut age existe
                        // dans la classe Ancetre
  }
}
```

Attention : une classe n'hérite pas des constructeurs de la classe qu'elle étend. Il faut nécessairement toujours définir dans une classe tous les constructeurs dont elle a besoin, même si ces derniers ne font que des appels aux constructeurs du père par `super(arguments)`.

Transtypage des objets



Implicite:	Explicite:	Impossible
Un carré	Une FormeGeometrique	Un Contrat
est	**peut être**	**n'est pas**
une FormeGeometrique	un Carre	une FormeGeometrique

Nous avons vu le transtypage des variables de types primitifs dans le précédent module. Le principe est de stocker dans une variable une valeur d'un autre type, qui sera "adaptée".

Le principe du transtypage des objets part du même principe : mettre la référence d'un objet d'un certain type dans une variable objet d'un autre type.

Nous allons voir qu'il existe trois possibilités de transtypage. Il peut être :

- Implicite.
- Explicite.
- Impossible.

Transtypage implicite

Comme nous l'avons vu, l'héritage permet de créer des objets ayant plusieurs types : Leur propre type et tous ceux des ancêtres.

Par exemple, un `Carre` est aussi un `Rectangle`, un `FormeGeometrique` et enfin un `Object`.

On peut donc considérer, implicitement, qu'un objet de type `Rectangle` est de type `Object`, donc peut être stocké dans une variable de ce type :

```
Object o= new Rectangle();
```

Ici, on crée un objet de type `Rectangle`, et on stocke sa référence dans une variable `Object`. Le transtypage est implicite : un `Rectangle` est bien un `Object`.

Transtypage explicite

Si un `Rectangle` est bien `FormeGeometrique`, le contraire n'est pas forcément vrai. Par exemple un `Cercle`, qui est aussi `FormeGeometrique`, n'est pas un `Rectangle`.

Supposons que l'on stocke dans une variable de type `FormeGeometrique` un objet de type `Rectangle` (transtypage implicite), et que l'on veuille récupérer cet objet (maintenant de type `FormeGeometrique`) dans une variable `Rectangle` : il faudra alors spécifier que cet objet `FormeGeometrique` est bien en `Rectangle`.

Exemple :

```
FormeGeometrique forme= new Rectangle();
Rectangle r= forme; // Cette ligne est incorrecte
```

Dans cet exemple, rien ne devrait s'opposer à ce que la variable r (de type `Rectangle`) ne reçoive la valeur de forme, qui contient un objet de type `Rectangle`. Pourtant, le compilateur n'acceptera pas une telle assignation, en effet, même si dans cet exemple forme contient bien un `Rectangle`, il pourrait tout aussi bien contenir autre chose, un `Cercle` par exemple!

Il est donc nécessaire de "forcer" l'assignation par un `cast` :

```
Rectangle r= (Rectangle) forme;
```

La syntaxe est similaire à ce que nous avons déjà vu avec les types primitifs.

Ce "casting" est de la responsabilité du programmeur, et peut provoquer des erreurs s'il est mal utilisé. Par exemple :

```
FormeGeometrique forme= new Cercle();
Rectangle r= (Rectangle)forme();
```

Syntaxiquement, ces deux lignes sont justes. Le compilateur ne détectera pas d'erreur, et pourtant, dans la logique il y a un problème : on essaie de faire passer un objet `Cercle` pour un `Rectangle`.

Ce transtypage est impossible : une erreur surviendra lors de l'exécution de l'application : une exception de type `ClassCastException` sera lancée (le prochain chapitre parlera des exceptions).

Transtypage impossible

Le transtypage est impossible lorsque les types ne sont pas compatibles, c'est à dire lorsqu'ils n'ont pas de lien de parenté par héritage.

Exemples :

```
Carre c= (Carre) new String( "Hello");
String s= (String) new Rectangle( 12, 4);
Color= (Color)new Carre( 34);
```

Une `String` n'est pas un `Carre`, un `Rectangle` n'est pas une `String`, et un `Carre` n'est pas une `Color`.

Dans ce cas, l'erreur est signalée au moment de la compilation. Même si on utilise un `cast` (comme sur les exemples ci-dessus), le compilateur ne sera pas d'accord : il vérifie l'existence d'un lien de parenté par héritage.

L'opérateur instanceof

Pour éviter les erreurs d'incompatibilités de types dans le transtypage, il est possible d'utiliser cet opérateur afin de tester si un objet est d'un certain type.

Il prend en argument de gauche un objet et en argument de droite le nom d'une classe.

Exemples :

```
Rectangle r= new Carre();
```

```
FormeGeometrique f= new Rectangle();
r instanceof Object              // Renvoie true
r instanceof FormeGeometrique    // Renvoie true
r instanceof Rectangle           // Renvoie true
r instanceof Carre               // Renvoie true
f instanceof Object              // Renvoie true
f instanceof FormeGeometrique    // Renvoie true
f instanceof Rectangle           // Renvoie true
f instanceof Carre               // Renvoie false!
```

Transtypage d'objets et invocations des méthodes

En Java, les méthodes sont toujours des fonctions virtuelles. C'est à dire que quel que soit le type de la variable contenant la référence de l'objet, la méthode invoquée sera celle de la classe réelle de l'objet.

Exemple :

```
Rectangle r= new Carre( 34);
r.paint(); // Exécute la méthode paint de la classe Carre et
           // non pas celle de la classe Rectangle (sauf si
           // cette méthode n'a pas été définie dans la classe
           // Carre)
```

Attention, pour les propriétés c'est différent, ce sont celles accessibles dans la classe correspondant au type de la variable qui sont utilisées.

Exemple :

```
public class TestHeritage {
  String s= "TestHeritage";

  public static void main( String [] args) {
    ClasseFils cf= new ClasseFils();
    System.out.println( "cf-> "+cf.s);
    TestHeritage th= cf;
    System.out.println( "th-> "+th.s);
  }
}
class ClasseFils extends TestHeritage{
  String s= "ClasseFils";
}
```

Cet exemple rend :

```
cf-> ClasseFils
th-> TestHeritage
```

Ce qui peut surprendre puisqu'il s'agit du même objet!

Tableaux d'objets

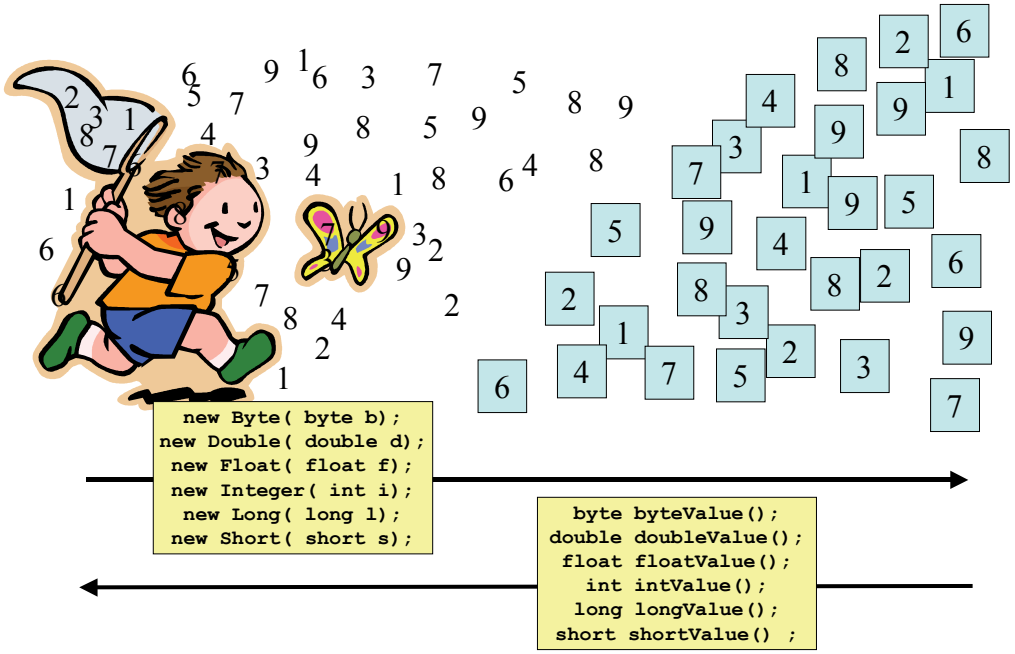
On peut stocker des objets de n'importe quel type dans un tableau, si le type de ses cellules est `Object`.

Exemple :

```
String s= "Toto";
Date d= new Date();
Object[] to= new Object[2];
to[0]= s;
to[1]= d;
// Dans l'autre sens, il faut caster:
s= (String) to[0];
d= (Date)to[1];
d= (Date)to[0]; // Ca passe à la compile,
                // mais ça casse à l'exécution
```

Lorsque l'on utilisera des collections d'objets, on vérifiera donc bien le type des objets avant de les utiliser.

Les objets numériques



En Java, les nombres sont dans des variables, qui sont de simples cellules mémoire, mais pas des objets. On appelle cela des types primitifs.

Cela ne pose pas de problème en soi, lorsqu'on les utilise simplement pour faire des opérations, ou des passages de paramètres.

Mais, comme nous les verrons plus loin (dans les collections d'objets ou dans la sérialisation), cela en pose un lorsque l'on a besoin de manipuler une variable en tant qu'objet et pas seulement en tant que valeur.

Pour manipuler un entier (**`int`**) ou un réel (**`float`**) ou autre type primitif, comme s'il s'agissait d'un objet, on dispose dans java de types numériques, aussi appelés « wrappers ».

Les types Wrapper

Chaque type primitif a son pendant en objet, qui hérite de la classe Number :

byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Remarque :
Le type objet porte le même nom que le type primitif, mais commence par une majuscule, à l'exception du type primitif **`int`** dont le type objet est **`Integer`**.

Pour créer un objet numérique, le constructeur prend en paramètre la valeur du numérique qu'il contiendra. Exemple :

```
public Integer( int n) ;
public Long( long l) ;
public Double( double d);
etc…
```

A noter aussi un constructeur qui prend une chaîne de caractères qu'il décodera :

```
public Integer( String s) ;
public Long( String s) ;
etc…
```

Pour récupérer la valeur du numérique qu'il contient, les méthodes ci-dessous son implémentées dans chacun des types.

```
byte byteValue();
double doubleValue();
float floatValue();
int intValue();
long longValue();
short shortValue() ;
```

L'usage des ces méthodes peut engendrer des conversions qui peuvent provoquer des troncatures ou la perte de la partie décimale lors du passage dans un entier, comme par exemple :

```
Double f= 234.643;
int i= f.intValue(); // i contiendra la valeur entière 234
```

On notera enfin les principales méthodes communes à tous ces types numériques :

```
String toString() ;
int compareTo( TypeDeLObjet objetIntegerCompare);
boolean equals( Object o);
static TypeDeLObjet valueOf( String s);
```

D'autres méthodes sont à découvrir, particulières à chaque type, comme par exemple la conversion en héxadécimal pour les entiers, la représentation en divers formats à virgule flottante pour les réels, etc. Veuillez vous reporter à la documentation des classes.

L'autoboxing

On a découvert cette possibilité dans le langage de programmation C# de Microsoft. Miracle de la concurrence : Sun l'a ajouté (JSR 201) dans la version 5 de Java !

Jusqu'à présent, les lignes suivantes étaient incorrectes :

```
int i= new Integer( 34) ;
Integer n= 32;
```

En effet, i n'est pas un objet et n'est pas un entier.

Cela devait être codé comme suit :

```
int i= (new Integer(34)).intValue() ;
Integer n= new Integer(32);
```

Grâce à l'autoboxing, lorsque l'on cherche à mettre le contenu d'une variable primitive dans un objet ou l'inverse, les transformations et transtypages sont totalement automatiques. Exemples :

```
Long l= 42L ;
Integer i= 34 ;
Long total= i+l;
```

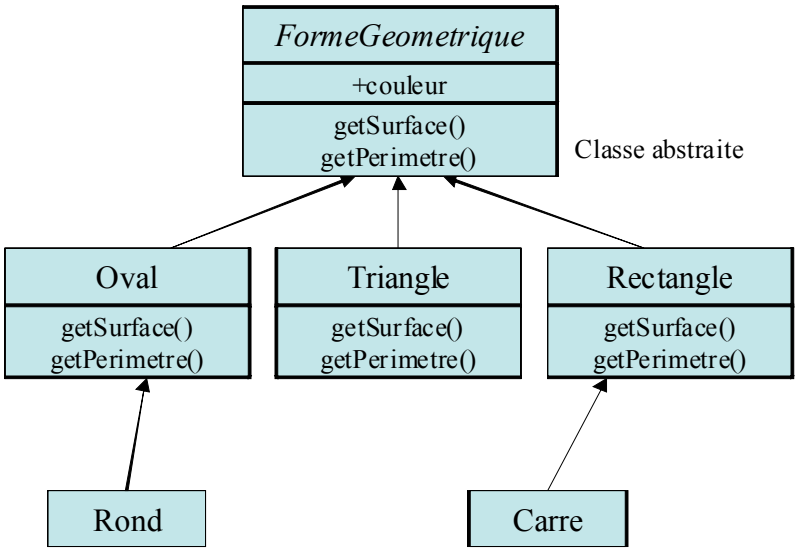
Remarque :
Si l'on cherche à autoboxer un objet null, alors c'est la valeur 0 qui est prise par défaut.

Exemple :

```
Integer i ; // Objet i à null
int n= i+3; // n prend la valeur 0+3
```

Il n'y a pas envoi d'une NullPointerException (vu au chapitre suivant), ce qui n'est pas très correct du point de vue Java, car un objet non initialisé cache souvent un bug.

Les classes abstraites



Ce sont des classes dont certaines méthodes ne sont pas implémentées.

Le rôle de ces classes est de définir des concepts au travers de méthodes qui seront obligatoirement implémentées par les héritiers.

On ne peut pas instancier une classe abstraite, puisque certaines méthodes n'ont pas de code et ne peuvent donc pas être appelées.

Les classes qui héritent d'une classe abstraite sont elles même abstraites (non instanciables) sauf si elles implémentent toutes les méthodes abstraites.

Pour créer une classe instanciable qui hérite d'une classe abstraite, on doit donc obligatoirement implémenter du code pour chaque méthode abstraite. C'est une sorte d'engagement ou de contrat.

L'exemple du transparent illustre bien ce concept avec les classes géométriques.

La classe `FormeGeometrique` représente le concept de forme géométrique. On y trouve la propriété couleur et les méthodes `getSurface`() et `getPerimetre`().

Ces méthodes calculent la surface et le périmètre de la forme. Elles ne peuvent pas être implémentées dans la classe `FormeGeometrique` : elle est trop abstraite.

Par contre, elles seront implémentées dans les formes géométriques concrètes : `Oval`, `Triangle` et `Rectangle`.

Remarque :
En notation UML, les classes abstraites sont notées en italique.

Les classes abstraites en Java

En Java, les classes abstraites sont spécifiées par le modificateur abstract.

Les méthodes abstraites sont simplement déclarées par leurs prototypes.

Exemple :

```
public abstract class FormeGeometrique {
  Color couleur;
  // Méthodes abstraites:
  public double getSurface();
  public double getPerimetre();
  // Méthodes concrètes:
  public void setCouleur( Color coul) {
    couleur= coul;
  }
  public Color getCouleur() {
    return couleur;
  }
}
```

Ceci est la définition du concept de la forme géométrique. Une forme doit retourner sa surface, son périmètre et sa couleur, on doit aussi pouvoir modifier sa couleur.

Les calculs de sa surface et de son périmètre dépendent de sa nature. Ils seront donc implémentés en fonction de la nature de la forme qui sera définie par une classe héritant de `FormeGeometrique`.

Remarque :
Il est nécessaire de spécifier qu'une classe est abstraite à l'aide du mot clé `abstract`, sinon le compilateur refusera de compiler cette classe. C'est ce modificateur qui interdit l'instanciation avec l'opérateur `new`.

Voici quelques implémentations de `FormeGeometrique` :

```
public class Rectangle extends FormeGeometrique {
  double longueur, largeur;
  public Rectangle( double longueur, double largeur) {
    this.longueur= longueur;
    this.largeur= largeur;
  }
  public double getSurface() {
    return longueur*largeur;
  }
  public double getPerimetre() {
    return 2*(longueur + largeur);
  }
}
// Un carré est un rectangle dont la hauteur et la largeur
// sont identiques.
public class Carre extends Rectangle {
  // Tout est géré dans Rectangle que l'on étend.
  // Le constructeur prend le côté du carré en argument
  // et appelle le constructeur de Rectangle qui prend
```

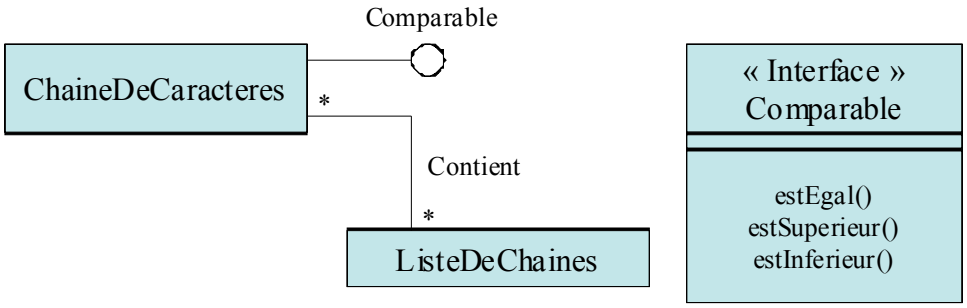
```
  // en argument la longueur et la largeur du rectangle
  public Carre(  double cote) {
    super( cote, cote);
  }
}
```

Dans cet exemple, `Carre` hérite de `Rectangle`, qui hérite de `FormeGeometrique`. `Carre` est donc bien de type `FormeGeometrique`.

```
public class Cercle extends FormeGeometrique {
  double rayon;
  public static final double PI= 3.141592;
  public Cercle( double rayon) {
    this.rayon= rayon;
  }
  public getSurface() {
    return PI*rayon*rayon;
  }
  public getPerimetre() {
    return PI*rayon*2;
  }
}
```

On voit dans cet exemple l'utilisation d'une variable statique : `PI`.

Les interfaces



L'interface d'un objet représente sa partie visible, c'est-à-dire principalement ses méthodes publiques.

Il est possible en Java de définir, dans un fichier, une interface sans implémentation, donc simplement une liste de signatures de méthodes.

Puis il sera possible de déclarer dans une classe que l'on implémente cette interface, c'est-à-dire que l'on implémente toutes les méthodes qui y sont déclarées.

On ne peut pas parler d'héritage dans le cas des interfaces, car pour qu'il y ait héritage il faut qu'il y ait quelque chose à hériter, or les interfaces ne sont que des coquilles vides. Mais c'est en tous les cas l'obligation, pour pouvoir compiler la classe, d'implémenter **toutes** les méthodes de l'interface en les redéfinissant.

Ce principe permet de définir un comportement abstrait, qui sera implémenté par plusieurs classes.

Exemple :

```
public interface Comparable {
  boolean estEgal( Object o);
  boolean estSuperieur( Object o);
  boolean estInferieur( Object o);
}
```

Une interface se crée dans un fichier portant son nom (ici ce sera Comparable.java) et se déclare comme une classe, à l'exception du mot clé `interface` qui prend la place du mot `class`.

Dans notre exemple, cette interface permet de définir le concept de comparaison. Elle permettra de comparer, donc par exemple de trier, des objets de types de classes qui l'implémentent.

Prenons l'exemple de notre classe `Carre`. Nous allons implémenter dans cette classe notre interface `Comparable`, ce qui nous permettra de comparer deux objets de type `Carre`.

La logique de comparaison sera propre à la logique métier des classes qui l'implémentent. Par exemple, nous supposerons que deux carrés sont égaux lorsqu'ils ont la même couleur et la même dimension.

```
public Carre extends Rectangle implements Comparable{
  public Carre(  double cote) {
    super( cote, cote);
  }
  boolean estEgal( Object o) {
    Carre c= (Carre)o;
    if( Couleur.equals( c.couleur) && longueur==o.longueur)
      return true;
    else
      return false;
  }
}
```

Examinons le code :

- La première ligne déclare la classe `Carre` et spécifie qu'elle implémente `Comparable`. Une classe peut implémenter autant d'interfaces qu'on le souhaite. Elles seront spécifiées toutes à ce niveau, séparées par des virgules.
- On retrouve le code de notre `Carre` (qui se limite à un constructeur qui fait appel au père `Rectangle`), auquel s'ajoute l'implémentation de la méthode `estEgal`.
- La première chose que fait notre méthode est de déclarer un objet `c` de type `Carre`, et de lui donner comme valeur celle de l'objet `o` de type `Object`, ce qui explique le typage en `Carre` de `o` en spécifiant entre parenthèses la classe `Carre` devant `o`. En fait, la méthode `estEgal` vient d'une interface qui se veut générique (comparaison de tous types d'objets). C'est la raison pour laquelle nous utilisons le type `Object`. Il est le plus générique puisqu'au sommet de la hiérarchie des classes : tout objet, quel que soit son type, est de type `Object`.
- Enfin, la comparaison passe par une comparaison, à la fois de la longueur, mais aussi de la couleur des carrés.

Dans cet exemple, le fichier `Carre.java` ne pourra pas être compilé, car les méthodes `estSuperieur` et `estInferieur` n'ont pas été implémentées. Le compilateur lancera siumplement le message d'erreur signalant que la classe `Carre` est abstraite, et doit de ce fait être déclarée comme telle.

L'interface est un contrat

Nous voyons bien ici la notion de contrat. Si une classe implémente `Comparable`, elle s'engage à implémenter toutes ses méthodes.

On appréciera le nommage des interfaces par un nom se terminant par "able" (`Comparable`, `Triable` ou `Sortable`, `Affichable`, `Imprimable`, `Calculable`, `Serializable`, `Stockable`, etc.)

Héritage entre les interfaces

Une interface ne peut implémenter une autre interface, bien sûr, puisqu'il n'y a pas d'implémentation dans une interface.

Par contre, une interface peut hériter d'une autre interface, et même de plusieurs autres interfaces en Java. L'héritage multiple étant ici autorisé, car il n'y a pas de problème de conflits d'implémentation, et pour cause!

Lorsque dans une classe on implémente une interface qui hérite d'autres interfaces, nous devons évidemment implémenter toutes les méthodes de toutes ces interfaces. Il faut donc toutes les examiner, c'est la raison pour laquelle il vaut mieux ne pas trop utiliser l'héritage dans les interfaces.

Les types interfaces

Un objet qui implémente une interface est du type de cette interface.

Ainsi, bien que l'on ne puisse pas créer d'objet du type d'une interface, rien ne nous empêche de déclarer une variable du type d'une interface, de l'initialiser avec un objet qui implémente cette interface.

Exemple :

```
Comparable c= new Carre( 25);
```

Attention, l'objet `c` étant de type `Comparable`, seules les méthodes de l'interface `Comparable` peuvent être invoquées, mais certainement pas celles de `Carre`, bien que cet objet soit un `Carre`.

Nous verrons tout au long de ce cours que cette notion est très importante en Java.

Les interfaces ont des propriétés

On ne trouve pas que des méthodes dans les interfaces. On peut aussi y déclarer des propriétés.

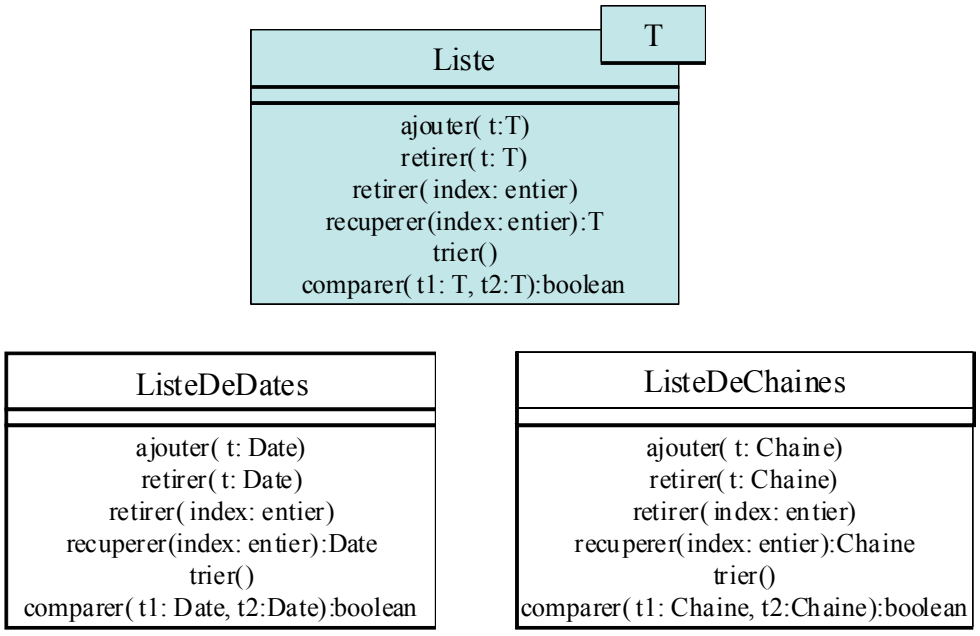
L'héritage multiple des interfaces peut générer des ambiguïtés au niveau du compilateur (si on implémente deux interfaces qui ont chacun une propriété de même nom). Pour éviter cet effet de bord, les propriétés des interfaces sont systématiquement statiques, même si on ne les déclare pas comme telles. De plus, leur accès ne peut se faire que par le nom de l'interface (propriétés de classes).

Conclusion

Les interfaces permettent :

- L'héritage multiple de concepts.
- L'engagement d'implémenter toutes les méthodes définies dans les interfaces (notion de contrat).
- Le transtypage d'objets incompatibles vers des types interfaces. On a la notion d'ensembles d'objets.

La généricité



La généricité permet de concevoir des classes génériques, véhiculant des concepts très sophistiqués, et qui seront implémentées de différentes manières suivant les types des objets qu'elles manipulent.

L'exemple le plus classique est la liste. Une telle classe doit être capable de manipuler des objets de différents types, avec des fonctions telles que l'insertion dans la liste, la suppression, la récupération, le tri, etc.

La généricité en Java

La généricité n'existe en Java qu'à partir de la version 5, avant, on s'en approchait avec le polymorphisme et les interfaces.

Prenons l'exemple typique de la liste. Nous voulons faire une classe `Liste` qui supporte le tri des éléments.

En Java, nous créerons simplement une classe qui va contenir des objets dont le type sera celui d'une interface (par exemple `Comparable`). Les objets que ma liste contiendra pourront être de n'importe quel type, pourvu que celui-ci implémente l'interface `Comparable`.

Exemple :

```
public class Liste {
  public void ajouter( Comparable element) {
    // etc.
  }
  public void retirer( Comparable element) {
    // etc.
  }
  public void retirer( int index) {
    // etc.
```

```
  }
  public Comparable recuperer( int index) {
    // etc.
  }
  public void trier() {
    // Appliquer un algorithme en s'appuyant sur les
    // méthodes de l'interface
  }
  public boolean comparer( Comparable e1, Comparable e2) {
    return e1.estEgal( e2);
  }
}
```

L'interface `Comparable` existe déjà dans Java :

```
public interface Comparable {
  public int compareTo(Object o);
}
```

La méthode `compareTo` retourne `0` si l'objet est égal (dans sa logique) à celui spécifié en argument, un nombre négatif s'il est plus petit, et un nombre positif s'il est plus grand.

Il restera simplement à implémenter cette interface dans toutes les classes d'objets que nous voudrons faire entrer dans la liste.

On notera que l'interface `Comparable` est implémentée dans la classe `String`. On peut donc faire entrer des chaînes de caractères dans notre `Liste`.

Exemple :

```
public static void main( String [] args) {
  Liste l= new Liste();
  // On fait entrer tous les arguments de la ligne de
  // commande dans la liste
  for( int n=0; n < args.length; n++)
    l.ajouter( args[n]);
  l.trier();
}
```

Les types génériques sont apparus dans la version 5 de Java. Ils sont surtout employés dans les classes de collections d'objets que nous verrons au chapitre 7.

Si nous reprenons notre exemple de Liste, il serait implémenté comme ceci :

```
public class Liste<A> {
  public void ajouter( A element) {
    // etc.
  }
  public void retirer( A element) {
    // etc.
```

```
  }
  public void retirer( int index) {
    // etc.
  }
  public Comparable recuperer( int index) {
    // etc.
  }
  public void trier() {
    // Appliquer un algorithme en s'appuyant sur les
    // méthodes de l'interface
  }
  public boolean comparer( A e1, A e2) {
    return e1.estEgal( e2);
  }
}
```

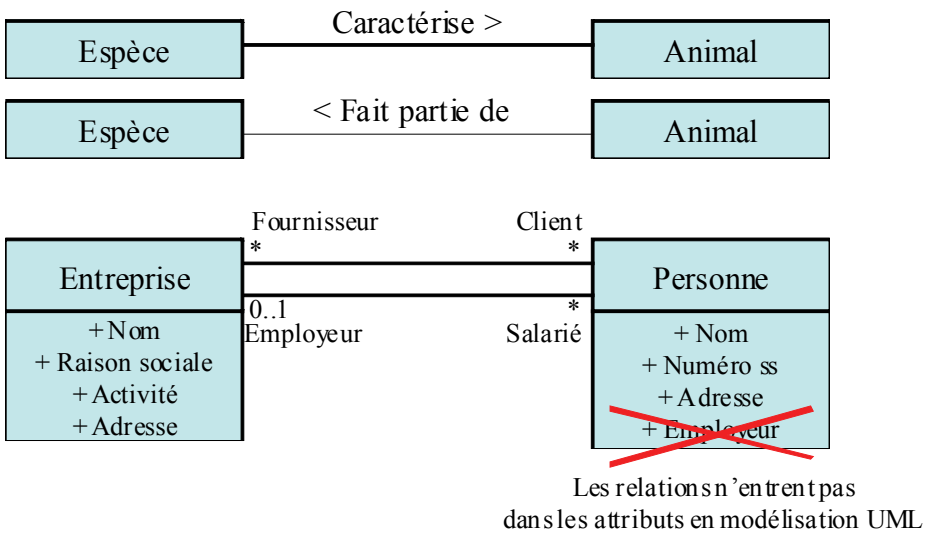
On voit donc que l'interface Comparable qui était employée avant est remplacé par un type A qui n'existe pas. Il n'existera que sous la forme qui sera choisie par le programme au moment de son utilisation.

Exemple :

```
public static void main( String [] args) {
  Liste<String> l= new Liste<String>();
  // On fait entrer tous les arguments de la ligne de
  // commande dans la liste
  for( int n=0; n < args.length; n++)
    l.ajouter( args[n]); // Vérifications à la compilation
                         // que c'est bien une String
  l.trier();
}
```

Dans cet exemple, on utilise la classe générique Liste avec des objets String. Dans la méthode `ajouter`, la vérification que c'est bien un objet String est faite lors de la compilation. Avant, c'était uniquement une vérification sur une interface qui pouvait être implémentée par des objets de types incompatibles.

Relations entre les classes



Les relations sont les liens qu'il peut y avoir entre différents objets.

Une relation peut être spécifiée en langage naturel. Par exemple, sur le premier exemple de la figure ci-dessus, on voit qu'une espèce caractérise un animal et qu'un animal fait partie d'une espèce.

Les relations peuvent avoir différents types de cardinalités (un vers un, un vers n, etc.), et on peut avoir plusieurs types de relations entre des objets.

Par exemple, le second cas du transparent montre qu'une personne peut être un client ou un salarié (ou les deux) pour une entreprise. Il y a donc là deux relations :

- Dans la première relation, une personne peut avoir un nombre indéfini d'entreprises fournisseurs et une entreprise un nombre indéfini de personnes clientes.
- Pour la seconde, une personne peut avoir zéro ou une entreprise employeur, et une entreprise peut avoir un nombre indéfini de personnes salariées.

On trouve différents types de cardinalités :

1	Un et un seul
0..1	Zéro ou un
*	De zéro à un nombre indéfini
1..*	De 1 à un nombre indéfini
n	Exactement n
m..n	De m à n

Pratiquement, les relations sont sous la forme de références vers un ou plusieurs autres objets dans une variable d'instance. Les relations sont donc des propriétés des objets.

Par exemple, les relations que l'on a entre `Entreprise` et `Personne` seront implémentées de la façon suivante :

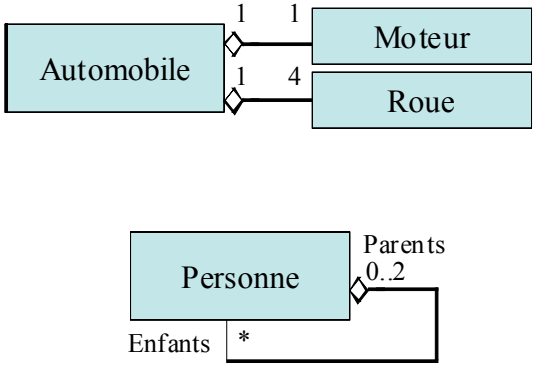
```
public class Entreprise {
  public String nom;
  punlic String raisonSociale;
  public String activite;
  public String adresse;
  // Une entreprise a 0, 1 ou plusieurs clients
  private Personne [] client;
  // Une entreprise a 0, 1 ou plusieurs salariés
  private Personne [] salarie;
}
```

```
public class Personne {
  public String nom;
  public String numeroSS;
  public String adresse;
  // Une personne a 0 ou 1 employeur (si c'est 0, la référence
  // de employeur est à null, sinon c'est la référence de
  // l'objet Employeur
  private Entreprise employeur;
  // Une personne a 0, 1 ou plusieurs fournisseurs
  private Entreprise [] fournisseur;
}
```

Remarque :
Dans la modélisation UML, les relations n'entrent pas dans les attributs, cela ferait double-emploi. Par exemple il ne sera pas correct d'ajouter dans la classe `Personne` l'attribut `employeur`.

Par contre, dans l'implémentation Java, `employeur` sera un attribut, dont le type dépendra du type de relation.

Les agrégations



L'agrégation est un type de relation. Son concept permet à un objet de posséder d'autres objets comme attributs. Il y a une notion **d'appartenance**.

Les deux exemples de la figure ci-dessus seront implémentés en Java de la façon suivante :

```
public class Automobile {
Moteur moteur;
Roue [] roues= new Roue[4];
}
```

Pour les roues, on utilise un tableau de quatre objets de type `Roue`. Nous reverrons dans le prochain chapitre la syntaxe de Java pour les tableaux.

```
public class Personne {
  Personne pere;
  Personne mere;
  Personne[] enfants;
}
```

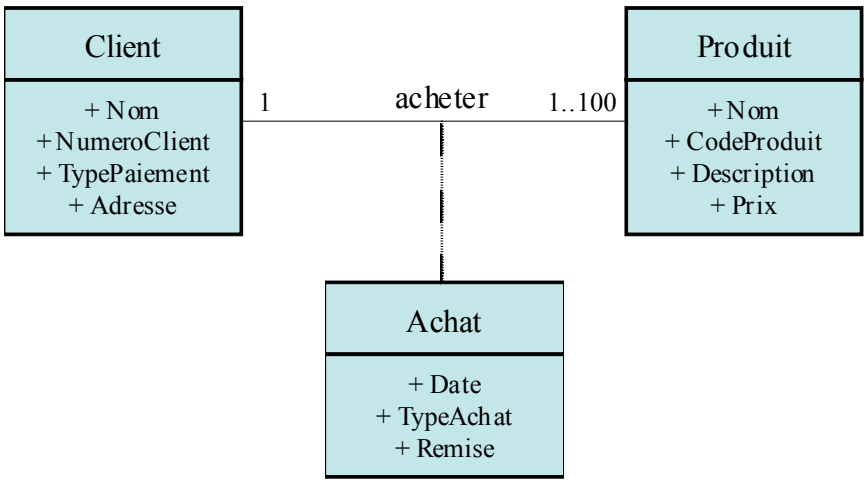
L'agrégation et la mémoire

Lorsqu'un objet n'est plus utilisé, il doit être détruit de la mémoire. S'il possède des objets agrégés, ces objets doivent aussi être détruits, sauf s'ils ont encore des relations avec d'autres objets non encore détruits.

On voit que la libération de la mémoire des objets inutiles peut vite devenir complexe, c'est aussi une source de bugs classique.

C'est la raison pour laquelle Java possède le Garbage Collector, qui prend en charge cette fonction, ce qui facilite bien la vie du développeur.

Les associations



L'association est une sorte d'index. Cela permet de définir une action entre deux classes.

Dans l'exemple de la figure ci-dessus, on voit bien les objets de type `Achat` qui seront effectués entre un objet de type `Client` et un objet de type `Produit`.

On voit aussi qu'un achat est composé d'un acheteur (l'objet `Client`) et d'au moins un produit (100 maximum).

L'implémentation en Java pourrait être la suivante :

```
public class Client {
  public String nom;
  public int numeroClient;
  public char typePaiement;
  public String adresse;
  // Les achats d'un client
  Achat[] achats;
}
```

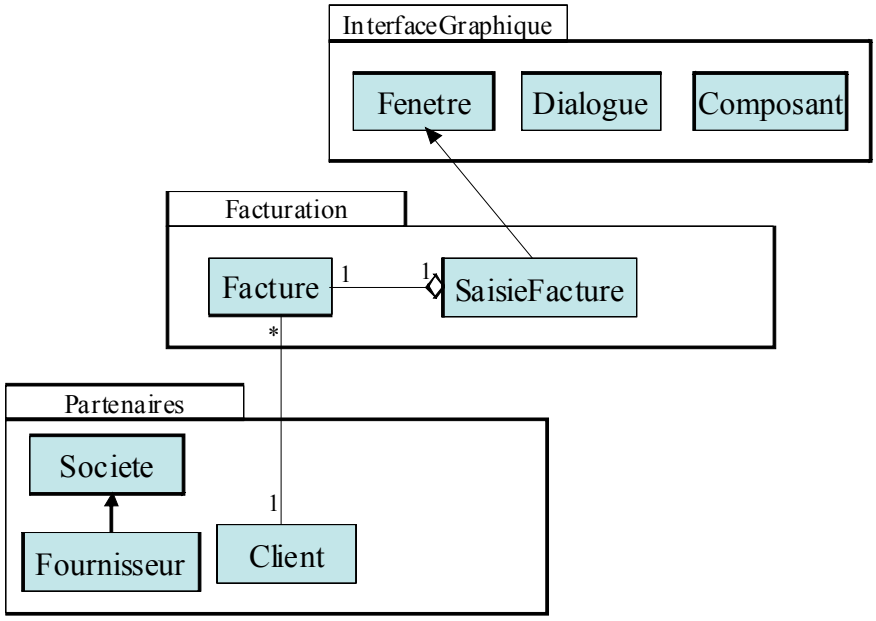
```
public class Produit {
  public String nom;
  public int codeProduit;
  public String description;
  public double prix;
}
public class Achat {
  public Date date;
```

```
  public char typeAchat;
  public float remise;
  public Client acheteur;
  // 100 produits maxi par achat
  public Produit [] produits= new Produit[100];
  // Compteur de produits dans le tableau
  int nombreProduits= 0;

  // Constructeur
  public Achat( Client client, Produit produit) {
    this.client= client;
    this.produit[0]= produit;
    nombreProduits++;
  }
  // Méthode pour ajouter un produit à un achat
  public void ajoutProduit( Produit produit) {
    this.produit[nombreProduits]= produit;
    nombreProduits++;
  }
}
```

On voit bien ici que l'association entre client et produit est faite dans le constructeur au moment de la création de l'objet `Achat`.

Les packages



Très rapidement, le développement d'applications va générer un nombre considérable de classes, et, comme dans Java, les classes sont stockées dans des fichiers séparés, le nombre important de fichiers va forcément nuire à l'organisation de projet.

Les packages permettent de rassembler des classes ayant des liens au niveau de l'organisation du logiciel, par exemple mettre ensemble les interfaces graphiques, puis dans un autre package les classes métier, etc.

Il faut noter que rien n'empêche de faire des héritages, des associations ou des agrégations entre classes de packages différents.

Certains langages comme Java, comme nous allons le voir juste après, prennent en compte le package dans la visibilité des classes et de leurs membres.

Créer des packages en Java

Les packages sont organisés de façon arborescente, exactement comme les répertoires des fichiers. C'est sur le gestionnaire de fichiers que Java s'appuie pour stocker les classes.

Pour créer un package, il suffit simplement de créer un répertoire, visible depuis l'une des entrées de la variable d'environnement CLASSPATH. Le nom de ce répertoire sera le nom du package.

Les répertoires devront être retrouvés, à la fois par le compilateur, mais aussi par l'interpréteur, c'est à dire la machine virtuelle. Ils utilisent tous les deux le CLASSPATH.

Exemple :

Nous allons créer une nouvelle entrée dans le CLASSPATH vers un répertoire appelé `monProjet` contenant les packages de notre projet :

```
C:\>md monProjet
C:\>set CLASSPATH=c:\monProjet;%CLASSPATH%
```

Dans ce répertoire, on crée un package appelé *outils* :

```
C:\>md monProjet\outils
```

Toutes les classes qui feront partie de ce package devront être stockées dans ce répertoire, de plus, elles devront être marquées comme faisant partie de ce package en mettant en première ligne de fichier java : `package nomDuPackage;`.

Exemple :

```
package outils;
public class Test{
  // etc.
}
```

Remarque :
Il est conseillé de toujours donner aux packages des noms en minuscules.

Les packages sont arborescents, on peut donc créer autant de sous packages que l'on veut dans les packages.

Ils seront stockés dans des sous-répertoires. Le nommage des sous-packages en java se fait en utilisant le point (**.**) comme séparateur.

Exemple :

On va créer le sous-package `graphique` dans le package `outils`.

```
C:\>md monProjet\outils\graphique
```

On stockera dans ce nouveau répertoire les classes en faisant partie, par exemple :

```
package outils.graphique;
public class Fenetre {
  // Etc.
}
```

Remarque :
On trouve de nombreuses classes outils disponibles sur Internet, payantes, mais aussi parfois gratuites. Ces outils sont dans des packages. Pour ne pas mélanger les packages provenant de différentes sources, Sun conseille de toujours créer ses propres packages dans un package contenant le nom de domaine de sa société, nom théoriquement unique.
Ainsi par exemple, si votre société possède le nom de domaine **acme.com**, vos packages devrons être dans le package : **com.acme** .

Utiliser des packages en Java

L'accès aux packages n'est pas automatique en Java. Il est nécessaire, pour chaque classe, de spécifier les packages dont elle a besoin.

Pour cela, on utilise l'instruction `import` en début de code.

Exemple :

```
package principal;
import outils.graphique.Fenetre;
import outils.*;
import java.util.*;
```

```
public class Application {
  // Etc.
}
```

Dans cet exemple, les trois lignes import permettent respectivement de :

- Utiliser la classe `Fenetre` du package `outils.graphique`.
- Utiliser toutes les classes du package `outils`.
- Utiliser toutes les classes du package `java.util`, qui fait partie de la machine virtuelle et contient divers utilitaires (dont nous en reparlerons plus tard). Le package `java` contient toutes les classes de Java organisées par domaine.

Remarque :
L'accès aux classes d'un package n'est pas récursif dans les sous-packages. Ainsi, par exemple, le deuxième import de notre exemple importe toutes les classes qui sont dans le répertoire outils, mais pas celles de ses sous répertoires (par exemple celles du répertoire `outils\graphique`).

Les imports statiques

Cela permet d'utiliser des propriétés (ou constantes) statiques de classes sans avoir à en hériter ou à les préfixer par leur classe. C'est pratique pour gagner du temps, mais attention aux risques de confusion…

La syntaxe consiste à déclarer un import avec le modificateur `static`. Cet import spécifie le chemin complet de la classe, suivi d'un point et du nom de la propriété statique ou d'une * si l'on veut cibler toutes les propriétés statiques de la classe.

Exemple :

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

public class Test{
  public static void main( String[] args) {
    out.println( "Hello plus simple");
  }
}
```

Dans cet exemple, je peux utiliser les membres statiques de la classe System (ici j'utilise la constante out qui pointe sur la sortie standard).

Les packages de Java

La machine virtuelle Java est livrée avec de nombreux packages dont nous allons étudier les principaux dans ce cours. Citons notamment :

java.lang	Toutes les classes de base du langage (String…).
java.util	Divers outils (dates, collections…).
java.io	Entrées sorties.
java.awt	Interface graphique.
java.awt.event	Gestion des événements de l'interface graphique.
java.sql	Gestion des bases de données (JDBC).
java.beans	Support des JavaBeans.

java.applet	Création d'applets.
java.net	Gestion du réseau.
java.security	Gestion de la sécurité.

Remarque :
Il n'est pas nécessaire d'importer le package `java.lang`, il est importé par défaut.

Les fichiers JAR

Une application est donc un ensemble de classes, disséminées dans des packages, parmi lesquels on trouvera peut-être des packages achetés à l'extérieur, ainsi que des packages propres à l'environnement d'exécution de l'application (serveur d'applications par exemple). De plus, l'application aura probablement besoin de divers fichiers de données pour fonctionner (configuration, images, scripts de bases de données, données de démarrage, etc.)

En phase de déploiement ou de distribution de l'application, il pourrait être intéressant de rassembler tout cela dans un seul fichier. C'est pour cela que l'on a inventé les fichiers JAR (Java ARchive).

Ces fichiers sont au format ZIP (On peut le consulter avec n'importe quel WinZIP du marché) dans lequel sont stockés, et donc éventuellement compressés, tous les fichiers d'une application (classes, fichiers XML, etc.)

Pour être utilisé, le fichier JAR n'a pas besoin d'être décompressé. La machine virtuelle Java sait aller trouver les classes et les autres fichiers directement dedans. Il suffit simplement d'ajouter le nom complet du fichier JAR comme une entrée du CLASSPATH.

Par exemple, si l'on a stocké le fichier `MonAppli.JAR` dans le répertoire c:\applications, il suffit de l'ajouter au CLASSPATH :

```
C:\>set CLASSPATH=c:\applications\MonAppli.JAR;%CLASSPATH%
```

Pour créer les fichiers JAR, un outil est mis à notre disposition dans le JDK : JAR.EXE.

Java Web Start

C'est le nom d'un produit proposé par Sun pour faciliter le déploiement des applications écrites en Java.

C'est une application cliente, qui permet le téléchargement et l'exécution d'applications Java à partir d'un serveur Web.

On dispose, grâce à cela, des facilités des architectures Web, tout en conservant la possibilité de bâtir des applications clientes complexes mais facilement distribuables.

Pour plus d'informations, se connecter à l'url :

http://java.sun.com/products/javawebstart/

Visibilité des membres en Java

Visibilité des membres	public	protected	private protected	friendly	private
La classe elle-même	X	X	X	X	X
Une classe héritière du même package	X	X	X	X	
Une classe du même package	X	X		X	
Une classe héritière d'un autre package	X	X	X		
Une classe d'un autre package	X				

Visibilité des classes	public	protected	friendly
Une classe héritière du même package	X	X	X
Une classe du même package	X	X	
Une classe héritière d'un autre package	X	X	X
Une classe d'un autre package	X		

En Java, on a 5 possibilités pour spécifier la visibilité d'un membre :

- `public` : le membre peut être accédé de n'importe quelle autre classe.
- `protected` : le membre peut être accédé par les classes de son propre package et par les classes qui en héritent, qu'elles soient ou non dans son package.
- `aucune spécification` (appelé « friendly ») : le membre peut être accédé uniquement par les classes de son propre package.
- `private protected` : le membre ne peut être accédé que par ses héritiers, qu'ils soient ou non dans son package.
- `private` : le membre ne peut pas être accédé d'une autre classe.

Visibilité des classes

On peut spécifier deux types de classes : `public` et « friendly ».

- La classe `public` sera visible d'absolument partout. Il faut, par ailleurs, que le source de la classe soit dans un fichier ayant le même nom qu'elle.
- La classe « friendly » ne sera visible qu'à l'intérieur du même package.

Remarque :
Une classe peut aussi être `private`, mais à condition d'être interne à une autre classe.

Visibilité des membres

L'usage des différents modificateurs de visibilité suit les règles suivantes :

- `private` : pour des membres utilisés exclusivement en interne de la classe. Leur invisibilité de l'extérieur améliore l'encapsulation de la classe.

- `public` : pour des membres dédiés au monde extérieurs (propriétés de configuration de l'objet, méthodes des services).
- `protected` : pour des membres à usage interne, mais qui doivent pouvoir être éventuellement manipulés par des méthodes de classes héritières.
- « friendly » : pour des membres dédiés uniquement au monde extérieur proche, c'est à dire les classes du même package.

Redéfinition des méthodes dans l'héritage

Les modificateurs des méthodes peuvent changer lorsqu'on les redéfinit, mais dans un seul sens : du plus restrictif au moins restrictif.

Par exemple, une méthode déclarée en `private` dans une classe pourra être redéfinie en `private`, `protected`, "friendly" ou `public` dans une classe fille, mais, une méthode déclarée en `public` ne pourra être redéfinie dans une fille que en `public`.

Exemple :

```
public class TestHeritage {
  private int methode1() { return 1;}
  public int methode2() { return 2;}
}

class ClasseFils extends TestHeritage{
  public int methode1() { return 1;}
  // methode2 ne peut qu'être public
  // private int methode2() { return 2;}
}
```

Les annotations

- **Directives spécifiées dans le code source des programmes**
  - @Override
  - @Depracated
  - @SupressWarnings
- **On peut créer ses propres annotations**
  - public @interface ToDo { … }
- **On peut créer son propre processeur d'annotations, qui sera invoqué à la compilation**
  - Génération de warnings ou d'erreurs
  - Génération de fichiers descripteurs

Les annotations permettent d'ajouter, dans les sources des programmes, des directives à l'attention des différentes cibles du code : Compilateur, Javadoc, interpréteur, analyseur de code…

Cette nouveauté de la version 5 est une évolution logique du langage de programmation. Déjà, on connaissait les annotations destinées à la documentation des classes (voir l'annexe de cet ouvrage consacrée à Javadoc), qui permettaient d'insérer des commentaires dans une forme normalisée afin de les exploiter dans un processus de documentation automatique.

On peut maintenant aller plus loin, et aussi annoter la compilation et l'exécution du code (runtime).

Tout l'intérêt des annotations est de permettre d'insérer directement dans le code des classes des directives qui étaient jusqu'à présent mises dans des fichiers de configuration séparés (fichiers descripteurs).

Fonctionnement des opérations

Les annotations sont associées à des classes ou à des membres. Elles précèdent le modifier (`public`, `static`…).

Elles sont indiquées par un mot clé qui commence toujours par une arobase.

Exemple :

```
@Deprecated public vieilleMethode () { // etc...
```

Les annotations standards de Java

La version 5 de Java contient 3 annotations standards :

```
@Override
@Deprecated
@SuppressWarnings
```

@Override

Cette annotation s'utilise pour les méthodes uniquement, elle permet de spécifier qu'une méthode redéfinit une méthode héritée.

Elle est utile afin d'éviter les erreurs de frappe, lorsque l'on redéfinit une méthode. En effet, lors de la redéfinition, si on écrit la méthode avec un nom légèrement différent, ce sera alors une nouvelle méthode, elle ne redéfinira pas l'ancienne.

Exemple :

```
public class MonObjet extends Object {
@override
public String toStirng() {
  return "C'est moi";
}
```

On voit dans cet exemple une faute de frappe dans la méthode **`toString`**, écrite par erreur **`toStirng`**. Cette erreur sera signalée lors de la compilation puisque cette méthode n'existe pas dans la classe **`Object`**.

@Deprecated

Elle permet de spécifier qu'une méthode est dépréciée. Toute invocation de cette méthode entraînera un warning au moment de la compilation.

Remarque :
Il ne faut pas confondre cette annotation avec celle de javadoc : **`@deprecated`** (Attention, la première lettre est une minuscule dans le cas du **`@deprecated`** de javadoc).

La différence est importante : Dans le cas de javadoc, c'est une simple déclaration qui sera intégrée à la documentation, pour signaler que cette méthode ne doit plus être utilisée, et qui permet de documenter par exemple la raison pour laquelle cette API est dépréciée.

Dans le cas de l'annotation standard, elle impactera simplement le compilateur : Cette annotation génèrera un warning lors de la compilation si le code invoque cette méthode.

Exemple :

```
public class Societe {

    /**
    * Retourne le numéro de SIRET pour la TVA.
    * @return Numéro SIRET pour la TVA.
    * @deprecated Il faut maintenant utiliser le numéro
    * de TVA intracommunautaire. Invoquer la méthode
    * getIntraTVA()
    */
    @Deprecated
```

```
    public int getSiretTVA() {
          return numeroSiret;
    }
}
```

@SuppressWarnings

Cette annotation est à destination du compilateur et permet de supprimer l'affichage des warnings spécifiés en argument.

Exemple :

```
@SuppressWarnings ("deprecated") public class
    ClasseQuiVaUtiliserDesMethodesDepreciees { // etc…
```

On voit ci-dessus le danger de cette annotation : On perd définitivement la trace, lors de la compilation, de l'utilisation d'APIs dépréciées par cette classe.

Remarque :
Cette annotation peut aussi prendre en argument un tableau de chaînes de caractères :

```
@SuppressWarnings( { "deprecated", "unchecked" } )
```

Créer ses propres annotations

Il est aussi possible de créer ses propres annotations.

C'est principalement à destination d'outils qui prendront le relai au moment de la construction du code.

Imaginons par exemple, dans le cadre de la mise en place d'un processus qualité, un mécanisme à base d'annotations qui permet, au moment de la construction de l'application, de signaler les portions de code non testées ou incomplètes.

Nous allons créer les 2 annotations suivantes :

- **`@ToTest`** permettra de spécifier ce qui doit être testé.
- **`@ToDo`** permettra, dans le code, de spécifier ce qui reste à faire.

Puis nous créerons le programme qui permettra de lire les classes et de signaler ce qui reste à faire et à tester.

La définition d'une annotation ressemble à celle d'une interface (on reconnaît l'annotation au caractère @ qui la différencie d'une interface.

Exemple :

```
public @interface ToTest {}
```

L'utilisation des annotations dans le code sera possible partout où l'on utilise les modifiers **`public`**, **`static`** ou **`final`** (interfaces, classes, méthodes, attributs).

Par convention, on fera toujours précéder l'annotation du modifier.

Exemple :

```
@ToTest public class NombresComplexes { // etc…
```

Une annotation peut contenir des déclarations de méthodes, qui doivent répondre aux caractéristiques suivantes :

- Elles n'ont pas d'argument.

- Elles n'ont pas de clause « **throws** ».
- Elles ne retournent que des types primitifs, des **String**, des **Class**, des annotations et des énumérations ou des tableaux de ces mêmes types.
- Elles peuvent avoir des valeurs par défaut.

Ces méthodes cachent en fait des attributs.

Exemple :

```
public @interface ToDo {
    int id() ;
    String date() ;
    String description() ;
}
```

Des valeurs par défaut peuvent être spécifiées pour chaque membre.

Exemple :

```
public @interface ToDo {
    int id() default 0;
    String date() ;
    String description() ;
}
```

L'utilisation de cette annotation se fera de la manière suivante :

```
    @ToDo(
        id=137,
        date= "10/6/2007",
        description= "Implémenter les nombres négatifs"
    ) public class NombresComplexes { // etc…
```

Remarque :
Une annotation sans contenu (comme dans l'exemple **@ToTest**) est appelée un marqueur.

Une annotation contenant seulement un élément dont le nom est **value** est appelée une valeur.

Exemple de valeur :

```
public @interface Auteur { String value(); }
```

Qui sera utilisée ainsi :

```
@Auteur ("Jérôme Bougeault") public class MaClasse { // etc…
```

Tableau d'annotations

Il n'est pas possible de spécifier plusieurs fois la même annotation. Par exemple, le code ci-dessous n'est pas correct :

```
    @ToDo(
        id=137,
        date= "10/6/2007",
        description= "Implémenter les nombres négatifs"
    )
    @ToDo(
        id=138,
        date= "10/6/2007",
        description= "Commenter les méthodes"
    ) public class NombresComplexes { // etc…
```

Pour remédier à cela, on peut faire des annotations contenant des tableaux d'annotations.

Exemple :

```
@interface TableauToDo {
    ToDo [] tableau();
}
```

qui serait utilisé ainsi :

```
    @TableauToDo ( tableau= {
    @ToDo(
        id=137,
        date= "10/6/2007",
        description= "Implémenter les nombres négatifs"
    ),
    @ToDo(
        id=138,
        date= "10/6/2007",
        description= "Commenter les méthodes"
    )}) public class NombresComplexes { // etc…
```

Les méta annotations

Quatre méta-annotations sont disponibles pour la création de ses propres annotations :

```
@Documented
@Inherit
@Retention
@Target
```

Leur utilisation nécessite l'inclusion du package **`java.lang.annotation`**.

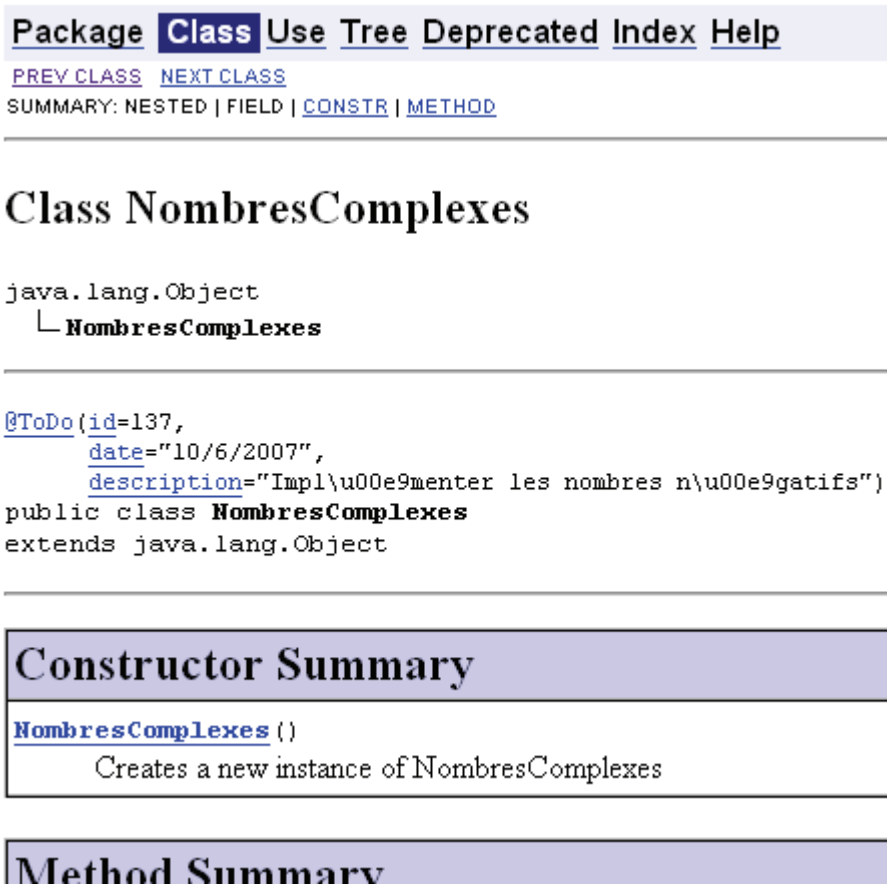
@Documented

Ne s'applique qu'à la déclaration d'une annotation. Permet de spécifier à l'outil javadoc que l'annotation doit être présente dans la documentation générée automatiquement.

Exemple :

```
import java.lang.annotation.* ;
@Documented public @interface ToDo {
    int id();
    String date();
    public String description();
}
```

Si l'on reprend l'exemple ci dessus de la classe **NombresComplexes**, le passage au javadoc donnera le résultat suivant :



On remarque que les accents, comme à l'accoutumée, passent très mal…

@Inherited

Par défaut, une annotation n'est pas héritée. Cela signifie qu'une classe héritant d'une superclasse ou un membre redéfinissant celui de sa superclasse n'hériteront pas des annotations de celle ci.

Cette méta-annotation permet de remédier à cela en faisant systématiquement hériter l'annotation spécifiée.

Exemple :

```
import java.lang.annotation.* ;
@Inherited @Documented public @interface ToDo {
    int id();
    String date();
    public String description();
}
```

@Retention

Elle permet de définir le périmètre de l'annotation à l'aide d'une des valeurs suivantes :

RetentionPolicy.SOURCE	L'annotation est à destination des outils qui exploitent les sources (compilateur, javadoc…). Elle ne sera pas écrite dans les fichiers compilés (.class).
RetentionPolicy.CLASS	L'annotation est à destination des outils qui exploitent les fichiers compilés, à l'exception de la JVM (par exemple les outils d'introspection). Elle est donc écrite dans les fichiers compilés.
RetentionPolicy.RUNTIME	Elle est écrite dans les fichiers compilés, mais ne sera utilisée que par la JVM au moment de l'exécution de l'application.

Exemple :

```
import java.lang.annotation.* ;
@Retention( RetentionPolicy.SOURCE) @Inherited @Documented
public @interface ToDo {
    int id();
    String date();
    public String description();
}
```

@Target

Enfin, on peut spécifier les types d'éléments sur lesquels on souhaite voir appliquer l'annotation à l'aide de cette méta-annotation :

ElementType.ANNOTATION_TYPE	sur une annotation.
ElementType.CONSTRUCTOR	sur un constructeur.
ElementType.FIELD	sur un attribut.
ElementType.LOCAL_VARIABLE	sur une variable locale.
ElementType.METHOD	sur une méthode.
ElementType.PACKAGE	sur un package.
ElementType.PARAMETER	sur le paramètre d'une méthode.
ElementType.TYPE	sur une classe, une interface ou une énumération.

Exemple :

```
import java.lang.annotation.* ;
@Target( ElementType.TYPE)
@Retention( RetentionPolicy.SOURCE) @Inherited @Documented
public @interface ToDo {
```

```
    int id();
    String date();
    public String description();
}
```

Remarque :
La méta-annotation **`@Target`** peut aussi recevoir un tableau en argument, si plusieurs types sont à prendre en compte. Exemple :

```
@Target( {ElementType.METHOD, ElementType.FIELD})
```

Annotation Processing Tool (APT)

Les annotations seront surtout destinées à renseigner, directement dans le code source, des informations susceptibles d'êtres utilisées par des outils autres que le compilateur. Par exemple des fichiers descripteurs (Descripteur de déploiement d'un EJB, classe BeanInfo d'un JavaBean…).

Cette génération automatisée de fichiers descripteurs pourra être effectuée par l'outil APT. Disponible à partir de Java 5, il fait partie des exécutables du JSDK.

La génération automatisée des fichiers descripteurs (ou autre) sera assurée par un code qu'il faudra développer, à partir de l'API « mirror ». Fournie en open source, et définie dans la JSR 269. Elle se trouve dans la package **`com.sun.mirror.apt`** qui se trouve dans le fichier **`tools.jar`** dans le répertoire **`lib`** du répertoire où est installé le JDK.

Pour l'utiliser avec NetBeans, choisir l'option menu « File / Project properties », puis l'option « Libraries », cliquer sur le bouton « Add Jar/Folder », et aller chercher le fichier tools.lib (par exemple en **`C:\jdk1.6.0\lib\tools.jar`**).

Il faudra aussi ajouter dans le processus de compilation l'invocation de la commande APT. C'est facilement réalisable si l'on utilise l'outil ANT.

Invocation d'un processeur depuis le compilateur

A partir de la version 6, l'outil APT n'est plus nécessaire. Il est possible d'invoquer un processeur directement dans la phase de compilation.

L'invocation s'effectue par l'option –processor du compilateur :

Exemple :

```
javac -processor ClasseProcesseur Programme.java
```

La classe processeur devra être présente dans le CLASSPATH au moment de la compilation.

Reprenons notre exemple d'annotation @ToDo, nous allons créer une classe processeur qui va simplement afficher, au moment de la compilation, le contenu de cette annotation à chaque fois qu'elle est trouvée dans le code à compiler.

Nous allons créer un package contenant à la fois la classe de l'annotation, et celle du processeur.

L'annotation **`ToDo`** ci-dessous fait partie du package **`mesannotations`**. Elle est utilisable soit pour une méthode, soit pour un constructeur.

```
package mesannotations;


import java.lang.annotation.*;
```

```
@Target( {ElementType.METHOD, ElementType.CONSTRUCTOR})
public @interface ToDo {
    String value();
}
```

Le processeur **ToDoProcessor** ci-dessous fera aussi partie du package **mesannotations**. Il se contente d'afficher l'annotation vers la console java, et d'envoyer un Warning.

```
package mesannotations;

import java.util.Set;
import javax.annotation.processing.*;
import javax.lang.model.SourceVersion;
import javax.lang.model.element.*;
import javax.tools.*;

@SupportedAnnotationTypes("*")
@SupportedSourceVersion( SourceVersion.RELEASE_6)
public class ToDoProcessor extends AbstractProcessor{
    /** Creates a new instance of ToDoProcessor */
    public ToDoProcessor() {
    }

    public boolean process(
            Set<? extends TypeElement> annotations,
            RoundEnvironment roundEnv) {
        for (Element e :
               roundEnv.getElementsAnnotatedWith(ToDo.class)) {
            if (e.getKind() != ElementKind.FIELD) {
                System.out.println(
                    "Traitement d'une annotation: "
                    +e.getAnnotation(ToDo.class)
                    +" dans "+e.getSimpleName());
                processingEnv.getMessager().printMessage(
                    Diagnostic.Kind.WARNING,
                    "ToDo dans: "+e.getEnclosingElement(), e);
                continue;
           }
        }
        return true;
    }
}
```

Enfin, l'utilisation de cette annotation se fera dans la classe **ClassePrincipale**, dans un autre package (**test**).

```
package test;
import mesannotations.ToDo;

public class ClassePrincipale {

    /** Creates a new instance of ClassePrincipale */
    @ToDo( "Le constructeur n'est pas encore implémenté")
    public ClassePrincipale() {
    }
    @ToDo ("Vérifier cette méthode")
    public static void main( String [] args) {
        System.out.println( "Exécution du programmme");
    }
}
```

Cette classe sera créée dans un autre projet, dont les paramètres de compilation prendront en compte l'utilisation du processeur (Aller dans les propriétés du projet « File / project properties ») :

- Aller dans l'option « Build / Compiling » et spécifier dans le champ « Additionnal Compiler Options » la ligne suivante :
  « **-processor mesannotations.ToDoProcessor** ».
- Aller dans l'option « Libraries » et cliquer sur « Add project » et choisir le projet contenant le processeur.

La compilation donnera l'affichage suivant :

```
Compiling 1 source file to C:\netbean\TestAPT\build\classes
Traitement d'une annotation: @mesannotations.ToDo(value=Le constructeur n'est pas encore implémenté) dans <init>
C:\netbean\TestAPT\src\test\ClassePrincipale.java:21: warning: ToDo dans: test.ClassePrincipale
    public ClassePrincipale() {
Traitement d'une annotation: @mesannotations.ToDo(value=Vérifier cette méthode) dans main
C:\netbean\TestAPT\src\test\ClassePrincipale.java:24: warning: ToDo dans: test.ClassePrincipale
    public static void main( String [] args) {
```

On voit dans cet exemple l'intérêt des annotations : permettre d'ajouter des actions de généralisation dans le processus de compilation du code.

Conclusion

Les annotations n'ont aucun effet sur la sémantique des programmes. Elles sont destinées à des outils lors de la construction du code.

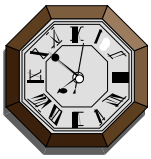
Cette nouveauté est assez peu utilisée dans le cadre de programmes J2SE, mais beaucoup plus utile pour la programmation serveur (J2EE), notamment pour les composants métiers (EJB) et plus généralement tous composants nécessitant des directives lors de leurs assemblages (Web Services…).

Mais il est probable que leur utilisation se développera aussi dans la version standard de Java, par exemple pour la génération des interfaces graphiques.

Atelier

Objectifs :

- **Mettre en pratique les concepts de l'objet (classes, objets, propriétés, méthodes, héritage, etc…)**
- **Bien comprendre les notions de polymorphisme et d'interface**



Durée minimum : 60 minutes.

Exercice 1 : Création d'une classe

Créer une classe `Date`, qui permettra de créer des objets représentant des dates.

Cette classe dispose de trois propriétés (de type `int`) :

- Jour.
- Mois.
- An.

Exercice 2 : Création d'objets

Dans la classe `Date` créée dans le premier exercice, implémenter une méthode `main`.

Dans cette méthode, créer trois objets `Date`, les initialiser à des dates différentes.

Exercice 3 : Variable de classe

Reprendre toujours la même classe `Date`, y définir une variable statique nommée `jourDeLAn`, de type `Date` et initialisée au premier janvier.

Cette variable nous permettra de vérifier si la date de l'objet est un jour de l'an (si le jour et le mois sont identiques à ceux de la variable `jourDeLAn`).

Exercice 4 : Méthode publique

Implémenter dans `Date` la méthode :

```
public void afficher()
```

Cette méthode affiche le contenu de l'objet date sous la forme jour/mois/an.

Si la date correspond au jour de l'an, afficher en plus le message "Bonne année".

Invoquer cette méthode dans le `main`, sur les trois objets `Date` créés.

Exercice 5 : Méthode privée

Implémenter une méthode interne (`private`) de vérification de la validité de la date (on se limitera à vérifier que le mois est compris entre 1 et 12 et le nombre de jours de chaque mois, en ne prenant pas en compte les années bissextiles : février fait toujours 28 jours).

```
private boolean verifDate();
```

Cette méthode rend false si la date est fausse.

Exercice 6 : Constructeur

Créer un `constructeur` qui prend en argument le jour, le mois et l'année. Utiliser ce constructeur dans le `main` pour créer les trois objets `Date`.

Créer aussi un constructeur sans argument.

Exercice 7 : Héritage

Créer une nouvelle classe : `Evenement`, qui hérite de `Date`.

La nouveauté dans cette classe est qu'elle possède une propriété supplémentaire qui contient le texte de l'événement (par exemple "anniversaire Toto", "Noël", etc.).

Créer le `constructeur` qui renseigne la date et le texte de l'événement :

```
public Evenement( int jour, int mois, int an, String texte);
```

Exercice 8 : Polymorphisme

Implémenter dans `Date` et `Evenement` les méthodes polymorphes :

```
String toString();
boolean equals( Object o);
```

`toString` renvoie une chaîne de caractères contenant :

- Pour la classe `Date` : la date au format jour/mois/an.
- Pour la classe `Evenement` : le texte de l'événement suivi de sa date au format jour/mois/an.

`equals` renvoie true si l'objet passé en argument est identique :

- Pour la classe `Date` : les jours, mois et ans sont identiques.
- Pour la classe `Evenement` : la date et le message sont identiques (pour ce dernier, s'appuyer sur la méthode `equals` de `String`).

Remarque :
La méthode polymorphe `equals` est un peu particulière.

Exercice 9 : Association

Récupérer l'algorithme du tri à bulles pour trier cette fois-ci des chaînes de caractères.

Voir dans la documentation du JDK les méthodes de la classe `String` qui permettent de comparer deux chaînes de caractères.

La nouvelle classe de tri (que l'on nommera `TriBulle2`) ne possèdera pas de méthode `static main`, mais un constructeur qui prendra en argument un tableau de `String`, une méthode pour déclencher le tri et une méthode pour afficher le contenu du tableau :

```
public class TriBulle2 {
  private String[] tableau; // Le tableau à trier, initialisé
                            // dans le constructeur ci-dessous
  // Constructeur
  public TriBulle2( String [] tableau) {
    // continuer ici…
  }
  public void trier() {
    // Trier le tableau ici…
  }
  public void afficher() {
    // Afficher chaque ligne du tableau ici
  }
}
```

Exercice 10 : Création d'interface

Et si nous voulions pouvoir trier n'importe quel type d'objet, comment faire ?

On peut utiliser l'interface `Comparable` déjà définie dans Java.

Attention à bien vérifier si les objets à comparer sont compatibles !

Reprendre le programme `TriBulle2`, le modifier pour qu'il puisse trier des objets `Comparable` au lieu d'objets `String`.

Remarque :
La classe `String` implémente l'interface `Comparable`. Avec cette nouvelle version de `TriBulle2`, on peut donc toujours trier des `String`.

Exercice 11 : Implémentation d'une interface

Tri de dates.

Pour trier des objets `Date` avec la classe de tri créée juste avant, il faut implémenter `Comparable` dans `Date`.

Exercice 12 : Création d'un package

Nous allons maintenant créer un package `outils` dans lequel nous allons mettre la classe de tri précédemment créée.

Questions/Réponses

Q. Quelle est la différence entre une variable locale, une variable globale et une variable d'instance ?

R. Les variables locales sont déclarées dans les méthodes, elles sont stockées dans la pile de programme, et ne sont donc accessible que à l'intérieur du bloc d'instruction de la méthode dans le contexte de son appel (deux appels successifs à la même méthode ne permettent pas au second appel de retrouver les données initialisées par le premier appel).

Les variables d'instance sont déclarées dans la classe, et sont stockées dans la mémoire globale. Mais elles sont propres à chaque objet. Deux objets n'ont pas les mêmes valeurs pour les mêmes variables. Lorsque l'on appelle deux fois la même méthode **d'un même objet**, les données initialisées par le premier appel sont récupérées dans le second appel.

Enfin, les variables globales sont des variables de classe, elles sont aussi déclarées dans la classe, mais avec le modificateur `static`. Elles sont stockées dans la mémoire globale et sont **partagées par toutes les instances** de la classe. Elles sont visibles uniquement dans les objets issus de la classe dans laquelle elles sont déclarées.

Q. Est-il possible d'appeler `super.super` ?

R. Non, en effet on sait que toute classe a un père (même une classe qui n'hérite de rien explicitement, héritera implicitement de `Object`). Par contre, elle ne possèdera pas forcément un grand-père.

Est-il autorisé d'hériter d'une classe abstraite mais de ne pas implémenter toutes les méthodes non implémentées de la classe abstraite ?

Oui, mais dans ce cas vous créez une autre classe abstraite, et il faudra la déclarer en `abstract`.

Q. Peut-on avoir, dans un même projet, des classes ayant exactement le même nom ?

R. Oui, à condition que ces classes soient dans des packages différents. Si vous souhaitez utiliser ces classes homonymes, simultanément dans une même classe de votre programme, il y a alors une ambiguïté qui ne peut être levée que en spécifiant ces classes par leur chemin complet. Cela est plutôt astreignant, ce qui expliquera que l'on évitera l'usage de noms identiques pour des classes différentes.

Cela existe, malgré tout, dans les classes de Java, comme par exemple les classes `java.util.Date` et `java.sql.Date`, qui représentent respectivement une date Java et une date SQL.

Q. Peut-on avoir un package dans deux entrées du CLASSPATH ?

R. Oui, les fichiers sont vus dans l'ordre des entrées dans cette variable d'environnement. Attention aux problèmes de conflits de version !

Q. Peut-on avoir une méthode avec un nombre d'arguments variable de types différents ?

R. Non, par contre si vous définissez que le type des arguments de votre méthode est Object, alors ce peut être des types différents, puisque tous les types objet descendent de Object. Attention à bien reconnaître les types des objets reçus avant de les « caster » pour les manipuler…

Q. Peut-on déclarer un constructeur `private`, et si oui à quoi cela peut-il servir ?

R. La réponse est oui. Un exemple d'utilisation pourrait être dans le cas d'une classe dont une seule instance doit être faite dans la JVM (par exemple un driver). Dans ce cas on déclarera le constructeur en `private` pour éviter qu'on ne l'utilise, mais il sera appelé depuis une méthode statique.

Exemple :

```
static ClasseAUneInstance uneSeuleInstance;
private ClasseAUneInstance() { // Constructeur private
}
// La méthode qui n'instanciera qu'une seule fois
// et retournera toujours l'instance
public static ClasseAUneInstance getInstance() {
  if( uneSeuleInstance == null)
    ClasseAUneInstance= new ClasseAUneInstance();
  return uneSeuleInstance;
}
```

Q. Peut-on accéder à un membre statique dans une classe abstraite ?

R. Oui, rien n'empêche d'accéder à une propriété statique, ou d'invoquer une méthode statique dans une classe abstraite.