Cours de Java 2

Sommaire

Avertissement : Le présent cours fait référence à la version 1.2 de Java, aussi appelé Java 2.

Chapitre 1 – Les bases du langage Java

Les données

Les primitives

Les concepteurs de *Java* ont doté ce langage d'une série d'éléments particuliers appelés *primitives*. Ces éléments ressemble à des objets, mais ne sont pas des objets ! Ils sont créés de façon différente, et sont également manipulés en mémoire de façon différente. Cependant il peuvent être *enveloppés* dans des objets spécialement conçu à cet effet, et appelés *enveloppeurs (wrappers)*.

?? **Table des primitives**

primitive	***étendue***	***taille***	***enveloppeur***
char	*0 à 65 535*	*16 bits*	*Character*
byte	*-128 à +127*	*8 bits*	*Byte*
short	*-32 768 à +32 767*	*16 bits*	*Short*
int	*- 2 147 483 648 à +2 147 483 647*	*32 bits*	*Integer*
long	*de -2^{63} à $+2^{63}-1$*	*64 bits*	*Long*
float	*de ? 1.4E-45 à ?3.40282347E38*	*32 bits*	*Float*
double	*de ? 4.9E-324 à ? 1.7976931348623157E308*	*64 bits*	*Double*
boolean	*true ou false*	*1 bit*	*Boolean*
void		*0 bit*	*Void*
	précision quelconque		*BigInteger*
	précision quelconque		*BigDecimal*

Les classes *BigInteger* et *BigDecimal* sont utilisés pour représenter respectivement des valeurs entières et décimales de précision quelconque. Il n'existe pas de primitives équivalente. Le type *char* sert à représenter les caractères, conformément au standard *UNICODE*. Il est le seul type numérique non signé ! Remarquons que le type *boolean* n'est pas un type numérique.

?? **Portabilité**

A l'inverse de ce qui se passe avec les autres langages, la taille des primitives est toujours la même en *Java*, et ce quelque soit l'environnement ou le type de la machine. On garantit ainsi la portabilité des programmes *Java*.

?? **Initialisation des primitives**

Les primitives doivent être déclarées et initialisées avant d'être utilisées. Une primitive non initialisée produira une erreur à la compilation : *« Variable may not have been initialized »*.

Remarquons que les primitives, lorsqu'elles sont employée comme membre de classe, possède des valeurs par défaut. Il n'est donc pas nécessaire de les initialiser !

primitive	***valeur par défaut***
boolean	*false*

primitive	***valeur par défaut***
char	*'\u0000' (null)*
byte	*(byte)0*
short	*(short)0*
int	*0*
long	*0L*
float	*0.0f*
double	*0.0d*

?? **Les valeurs littérales**

primitive	***syntaxe***
char	*'x'*
int	*5 (décimal), 05 (octal), 0x5 (hexadécimal)*
long	*5L, 05L, 0x5L*
float	*5.5f ou 5f ou 5.5E5f*
double	*5.5 ou 5.5d ou 5.5E5d ou 5.5E5*
boolean	*false ou true*

?? **Casting sur les primitives**

Les constantes

Java ne comporte pas de constantes à proprement parler. Il est cependant possible de simuler l'utilisation de constantes à l'aide du mot clé *final*. Une variable déclarée *final* ne peut plus être modifiée une fois qu'elle a été initialisée.

Lorsqu'un élément est déclaré *final*, le compilateur est à même d'optimiser le code compilé afin d'améliorer sa vitesse d'exécution.

Remarquons que les variables déclarées *final* peuvent être initialisées lors de l'exécution et non seulement lors de leur déclaration !

Les *handles*

définition…

?? ***final***

L'utilisation de final n'est pas réservé aux primitives. Un *handle* d'un objet peut parfaitement être déclaré *final*. Cependant, la contrainte ne s'applique alors qu'au handle, qui ne peut plus voir son affectation modifiée. L'objet, lui, reste modifiable.

Les chaînes de caractères

En Java, les chaînes de caractères sont des objets. Ce sont des instances de la classe *String*. Depuis les premiers langages de programmation, les chaînes de caractères ont posé des problèmes ! En effet, s'il est facile de définir différents types numériques de format fixe, les chaînes de caractères ne peuvent pas être représentées dans un format fixe car leur longueur peut varier de 0 à un nombre quelconque de caractères.

Java utilise une approche particulière. Les chaînes de caractères peuvent être initialisées à une valeur quelconque. Leur longueur est choisie en fonction de leur valeur d'initialisation. En revanche, leur contenu ne peut plus être modifié. En effet, la longueur des chaînes étant assurée de ne jamais varier, leur utilisation est très efficace en termes de performances.

Par ailleurs, il faut noter que Java dispose d'une autre classe, appelée *StringBuffer*, qui permet de gérer des chaînes dynamiques. Remarquons qu'il est également possible de traiter des instances de la classe *String* comme des chaînes dynamiques, sous certaines précautions.

?? **Chaînes littérales**

Les chaînes de caractères existent aussi sous forme littérale. Il suffit de placer la chaîne entre guillemets comme dans l'exemple suivant :

```
"Bonjour maman !"
```

Les chaînes littérales peuvent contenir des caractères spéciaux issues du type *char* :

caractères spéciaux	description
\b	backspace
\f	saut de page
\n	saut de ligne
\r	retour chariot
\t	tabulation horizontale
\\	\
\'	'
\"	"
\012	caractère en code octal
\uxxxx	caractère en code héxadécimal (unicode)

Les tableaux

Les tableaux Java sont des structures pouvant contenir un nombre fixe d'éléments de même nature. Il peut s'agir d'objet ou de primitives. Chaque élément est accessible grâce à un indice correspondant à sa position dans le tableau. Les tableaux de Java sont des objets, même lorsqu'ils contiennent des primitives.

?? **Déclaration**

?? **Initialisation**

Les vecteurs

Les collections

Les itérareurs

Les comparateurs

Les opérateurs

Les opérateurs Java sont voisins du C.

Description des principaux opérateurs

?? **Opérateur d'affectation**

	symbole	***description***	***arité***	***exemple***
opérateur d'affectation	=	*affectation*	*2*	*x = 2*

	–=	*soustraction et affectation*	*2*	*x–= 2*
	+=	*addition et affectation*	*2*	*x += 2*

On dispose du raccourci : *x = y = z = 2*.

?? **Opérateurs arithmétiques à deux opérandes**

	symbole	***description***	***arité***	***exemple***
opérateurs arithmétiques à deux opérandes	*–*	*soustraction*	*2*	*y – x*
	***	*multiplication*	*2*	*3 * x*
	/	*division*	*2*	*4 / 2*
	%	*modulo (reste de la division)*	*2*	*5 % 2*

Il n'existe pas en *Java* d'opérateur *d'exponentiation*. Pour effectuer une exponentiation, il convient d'utiliser la fonction *pow(double a, double b)* de la classe *java.lang.Math.*

La *division* des entiers fournit un résultat tronqué et non arrondi.

?? **Opérateurs à un opérande**

	symbole	***description***	***arité***	***exemple***
opérateurs à un opérande	*–*	*opposé*	*1*	*–x*
	++	*pré-incrémentation*	*1*	*++x*
	++	*post-incrémentation*	*1*	*x++*
	--	*pré-décrémentation*	*1*	*--x*
	--	*post-décrémentation*	*1*	*x--*

?? **Opérateurs relationnels**

	symbole	***description***	***arité***	***exemple***
opérateurs relationnels	*==*	*équivalent*	*2*	*x == 0*
	<	*plus petit que*	*2*	*x < 2*
	>	*plus grand que*	*2*	*x > 2*
	<=	*plus petit ou égal*	*2*	*x <= 3*
	>=	*plus grand ou égal*	*2*	*x >= 3*
	!=	*non équivalent*	*2*	*a != b*

Les notions d'équivalence et de non équivalence s'appliquent à toutes les primitives, ainsi qu'aux handles d'objets.

Il faut noter que l'équivalence appliquée aux handles d'objets concerne les handles, et non les objets eux-mêmes ! Deux handles sont équivalents s'ils pointent vers le même objet. Il ne s'agit donc pas d'objets égaux, mais d'un seul objet.

Pour tester si deux objets distincts (ou non) sont effectivement égaux, il convient d'utiliser la méthode *equals*.

Considérons le petit programme suivant :

```
Integer a = new Integer(100) ;
Integer b = new Integer(100) ;
Integer c = a ;

System.out.println(a == b) ;
System.out.println(a == c) ;
System.out.println(a.equals(b)) ;
```

La première impression affiche *false*, la deuxième *true* et la dernière *true*. Par ailleurs, on aurait pu tout aussi bien remplacer *a.equals(b)* par *b.equals(a)*.

?? **Méthode *equals***

En fait, la méthode *equals* appartient à la classe *Object* et toutes les autres classes en héritent. On donne ci-dessous sa définition initiale, qui compare les *handles* :

```
public equals(Object obj) {
   return (this == obj) ;
}
```

Dans cette définition, on constate que *equals* se comporte exactement comme ==. En revanche, dans la plus part des classes (comme *Integer*, voir l'exemple précédent) la méthode est redéfinie pour qu'elle compare le contenu des objets plutôt que leur *handles*.

Cette méthode est souvent problématique et il convient de consulter la documentation.

?? **Opérateurs logiques**

	symbole	***description***	***arité***	***exemple***
opérateurs logiques	*&&*	*et*	*2*	*a && b*
	||	*ou*	*2*	*a || b*
	!	*non*	*1*	*!a*

L'évaluation des expressions logiques est stoppées dès lors que le résultat est déterminé. L'évaluation partielle optimise le code mais peut avoir des effets indésirables. Une manière de forcer l'évaluation consiste à utiliser les opérateurs d'arithmétique binaire.

?? **Opérateurs d'arithmétique binaire**

Les opérateurs d'arithmétiques binaires agissent au niveau des bits de données, sans tenir comte de ce qu'ils représentent.

	symbole	***description***	***arité***	***exemple***
opérateurs d'arithmétique binaire	*&*	*et*	*2*	*a & b*
	|	*ou*	*2*	*a | b*
	^	*ou exclusif*	*2*	*a ^ b*
	~	*non*	*1*	*~a*
	<<	*décalage à gauche*	*2*	*a << 2*
	>>	*décalage à droite*	*2*	*b >> 2*
	>>>	*décalage à droite sans extension du signe*	*2*	*b >>> 2*

Rappelons que le bit de signe (des types entiers) est le bit de poids fort.

On peut utiliser les opérateurs d'arithmétique binaire avec des valeurs logiques, qui sont des valeurs sur *1* bit. L'intérêt de cette possibilité est que, s'agissant d'opérateurs arithmétiques, ils sont toujours évalués.

?? **L'opérateur à trois opérandes**

	symbole	***description***	***arité***	***exemple***
opérateur à trois opérandes	*? :*	*condition ? vrai : faux*	*3*	*y < 5 ? 4 * y : 2 * y*

Si *condition* est vrai, alors on retourne l'évaluation de l'expression *vrai*, sinon on retourne celle de *faux*.

?? **Opérateurs de *casting***

new

L'opérateur *new* permet d'instancier une classe, c'est-à-dire de créer une instance de cette classe.

instanceof

L'opérateur *instanceof* permet de tester un objet est une instance d'une classe donnée (ou de l'une de ses sous-classes). Il prend en paramètre à gauche un *handle*, et à droite un nom de *classe* ; il retourne un *boolean*.

```
String y = "bonjour" ;
boolean a = y instanceof String ;
```

Dans cet exemple, *a* vaut *true*.

L'opérateur *instanceof* ne permet de tester le type d'une primitive.

Il existe également une version dynamique de l'opérateur *instanceof*, sous la forme d'une méthode de la classe *class*.

L'opérateur + pour *String*

Les chaînes de caractères dispose de l'opérateur + qui permet de concaténer deux chaînes ; += est aussi valide pour les chaînes de caractères.

```
String x = "maman" ;
System.out.println("bonjour" + x) ;
```

Priorité des opérateurs

Les opérateurs par ordre de préséance décroissante sont les suivants :

Opérateurs	***Ordre d'évaluation***
Appel de méthode . [] () - +	*De gauche à droite*
++ -- ! ~ (cast) new	
** / %*	
+ -	
<< >> >>>	
< > <= >=	
== !=	
&	
^	
|	
&&	
||	
?:	
= op=	
,	

Les structures de contrôle

Les structures de contrôle sont presque les mêmes que celles utilisées en C ou en C++. On les rappelle ci-dessous en précisant les spécificités *Java*.

structure de contrôle	***syntaxe***	***commentaires***
retour d'une méthode	***return*** *;*	*cas d'une méthode void*

	return *(type de retour) ;*	
instruction conditionnelle ***if***	***if*** *(expression) instruction ;*	*les parenthèses sont obligatoire en Java autour de l'expression booléenne. De même le point-virgule est obligatoire en fin de bloc.*
	if (expression) {bloc d'instructions }	
instruction conditionnelle ***else***	***if*** *(expression) instruction ;* ***else*** *instruction ;*	
	if *(expression) {bloc d'instructions}* ***else*** *{bloc d'instructions}*	
	if *(expression) ;* ***else*** *{bloc d'instructions}*	*On ne peut pas donner une instruction vide après la condition if.*
instruction conditionnelle ***else if***	***if*** *(expression) {bloc d'instructions}* ***else if*** *{bloc d'instructions}* *…* ***else if*** *{bloc d'instructions}* ***else*** *{bloc d'instructions}*	*Il s'agit en fait d'une structure de contrôle à part entière écrite en deux mots !*
la boucle ***for***	***for****(initialisation ; test ; incrémentation) {bloc d'instructions}*	*la partie initialisation se compose d'une ou plusieurs initialisations (séparées par des virgules. La partie test est une expression booléenne. La partie incrémentation peut contenir plusieurs instruction séparées par des virgules.*
utilisation de ***break*** *et* ***continue*** *dans les boucles*	***break*** *;*	*Interruption de l'itération en cours et passage à l'instruction suivant la boucle.*
	continue *;*	*Interruption de l'itération en cours et retour au début de la boucle avec exécution de la partie incrémentation.*
l'instruction ***while***	***while*** *(expression booléenne) {bloc d'instructions}*	
l'instruction ***switch***	***switch*** *(variable)* ***case*** *valeur1 : instructions1 ;* ***case*** *valeur2 : instructions2 ;* *…* ***default*** *: instructions ;*	*Les bloc sont délimités par deux instructions* ***case*** *! Lorsqu'une égalité est trouvée, le bloc d'instruction correspondant est exécuté, ainsi que tous les blocs suivant ! Pour qu'un seul bloc ne soit exécuté, il faut utiliser explicitement l'instruction* ***break****.*

Mots clés

?? ***static***

Un élément déclaré *static* appartient à une classe et non à ses instances. Les objets instanciés à partir d'une classe ne possèdent pas les éléments de cette classe qui ont été déclaré *static*. Un seul élément existe pour la classe et il est partagé par toutes les instances. Cela ne limite en aucune façon l'accessibilité mais conditionne le résultat obtenu lors des accès. Les primitives, les objets et les méthodes peuvent être déclaré *static*.

?? ***final***

De nombreux langages de programmation, par exemple, font la différence entre les données dont la valeur peut être modifiée (les variables) et celles dont la valeur est fixe (les constantes). Java ne dispose pas de constantes. Ce n'est pas une limitation, car Java dispose d'un mécanisme beaucoup plus puissant, avec le modificateur *final*, qui permet non seulement d'utiliser une forme de constantes, mais également d'appliquer ce concept à d'autres éléments comme la méthode ou les classes.

?? ***synchronized***

Cf. Threads.

?? ***native***

Cf. Les méthodes.

?? ***transient***

Le mot clé *transient* s'applique aux variables d'instances (primitives et objets). Il indique que la variable correspondante est transitoire et que sa valeur ne doit pas être conservé lors des opérations de *sérialisation*. La sérialisation est une opération qui permet d'enregistrer un objet Java sur un disque, afin de le conserver pour une session ultérieure, ou de l'envoyer à travers un réseau. Lors de la sérialisation, tous les champs sont sauvegardés à l'exception de ceux déclarés *transient*.

?? ***volatile***

Le mot clé *volatile* s'applique aux variables pour indiquer qu'elles ne doivent pas être l'objet d'optimisation. En effet, le compilateur effectue certaines manipulations pour accélérer le traitement (utilisation des registres). Ces optimisations peuvent s'avérer problématiques dès lors que plusieurs processus utilisent la même variable, celle-ci risque de ne pas être à jour ! Il est donc nécessaire d'empêcher une telle optimisation.

?? ***abstract***

Le mot clé *abstract* peut être employé pour qualifier une classe ou une méthode.

Cf. Classes abstraites, Interface, Polymorphisme

Chapitre 2 – Concepts de base de la programmation orientée objet

Introduction

« Tout est objet ! »

Le principe fondamental d'un *langage orienté objet* est que le langage doit permettre d'exprimer la solution d'un problème à l'aide des éléments de ce problèmes. Par exemple, un programme traitant des images doit manipuler des structures de données représentant des images, et non leur traduction sous formes d'une suite de bits. De cette façon, on procède à une *abstraction*.

Java est l'aboutissement (pour le moment, du moins) de ce concept. Pour Java, l'univers du problème à traiter est constitué d'objets. Cette approche est la plus naturelle car elle correspond à notre façon d'appréhender l'univers, ce qui facilite la modélisation des problèmes.

Rappelons ici le deuxième et troisième précepte de *Descartes* extraits du *discours de la méthode* (1637) et qui vont dans ce sens :

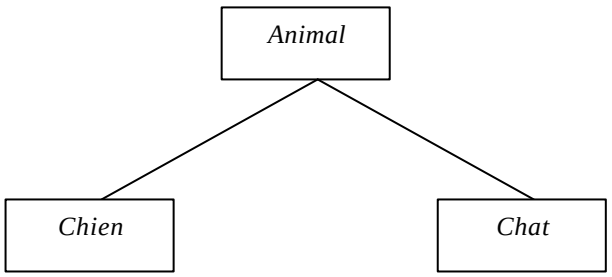
> *« Le deuxième, de diviser chacune des difficultés que j'examinerais , en autant de parcelles qu'il se pourrait, et qu'il serait requis pour les résoudre. »*
>
> *« Le troisième, de conduire par ordre mes pensées, en commençant par les objets les plus simples et les plus aisés à connaître, pour monter peu, comme par degrés, jusques à la connaissance des plus composés ; et supposant même l'ordre entre ceux qui ne se précèdent point naturellement les uns les autres. »*

Tout est donc *objet*. Il faut entendre par objet, *élément de l'univers* relatif au problème à traiter. Les objets appartiennent à des catégories appelées *classes*, qui divisent cet univers.

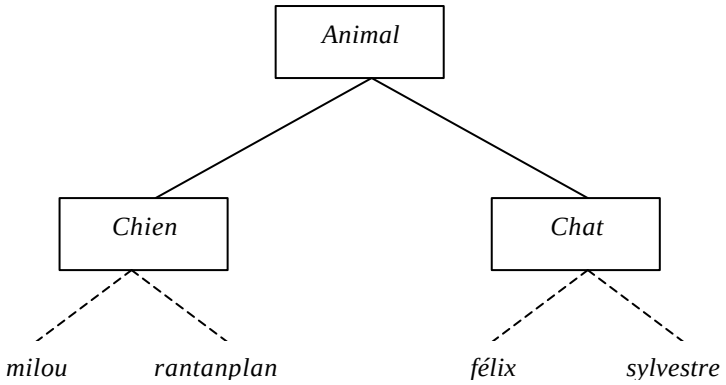
Illustration des concepts de classe et d'objet

Si notre problème concerne les animaux, nous pouvons créer une classe que nous appellerons *Animal*. Si nous devons considérer les chiens et les chats, nous créerons les trois classes dérivée de la classe *Animal* : les classes *Chien* et *Animal*. Peu importe que cette division ne soit pas pertinent dans l'univers réel, il suffit qu'elle le soit dans celui du problème à traiter.

Nous représenterons cette division de l'univers de la façon suivante :



Considérons à présent le modèle suivant :



Il est évident que le rapport entre *milou* et *Chien* n'est pas de la même nature que le rapport entre *Chien* et *Animal*. *Chien* est une *sous-classe* de *Animal*. Dans la terminologie de Java, on dit que *Chien* étend la classe *Animal*. En revanche *milou* n'est pas une classe. C'est un représentant de la classe *Chien*. Selon la terminologie Java, on dit que *milou* est une *instance* de la classe C*hien*.

Les classes

Définition

La classe regroupe la définition des *membres de classe*, c'est-à-dire :

- des *méthodes*, les opérations que l'on peut effectuer ;
- des *champs*, les variables que l'on peut traiter ;
- des *constructeurs*, qui permettent de créer des objets ;
- et encore d'autres choses plus particulières.

Plus précisément, une classe peut contenir des *variables (primitives ou objets)*, des *classes internes*, des *méthodes*, des *constructeurs*, et des *finaliseurs*.

La déclaration d'une classe se fait de la façon suivante :

```
[Modificateurs] class NomClasse

      {
            corps de la classe
      }
```

Le nom de la classe doit débuter par une majuscule.

Considérons l'exemple suivant :

```
Class Animal {

   // champs

   boolean vivant ;
   int âge ;

   // constructeurs

   Animal() {
   }

   // méthodes
```

```
    void viellit() {
          ++âge ;
    }

    void crie() {
    }

}
```

Les classes *final*

Une classe peut être déclarée *final*, dans un but de sécurité ou d'optimisation. Une classe *final* ne peut être étendue pour créer des sous-classes. Par conséquent, ses méthodes ne peuvent pas être redéfinies et leur accès peut donc se faire sans recherche dynamique.

Une classe final ne peut pas être clonée.

Les classes internes

Une classe Java peut contenir, outre des primitives, des objets (du moins leurs références) et des définitions de méthodes, des définitions de classe. Nous allons maintenant nous intéresser de plus près à cette possibilité.

?? **Plusieurs classes dans un même fichier**

Il arrive fréquemment que certaines classes ne soient utilisées que par une seule autre classe. Considérons l'exemple suivant :

```
Class Animal {

    // champs

    boolean vivant ;
    int âge ;
    Coordonnées position ;

    // constructeurs

    Animal() {
          position = new Coordonnées() ;
          }

    ...

    }

Class Coordonnées {

    // champs

    int x = 0;
    int y = 0;

    ...

    }
```

Toutes ces classes sont définies dans le même fichier, ce qui convient dans le cadre de la démonstration mais certainement pas pour la pratique courante de la programmation efficace. Chacune de ces classes pourrait être définie séparément dans un fichier et affectée à un package.

Lors de la compilation du précédent fichier *Animal.java*, le compilateur produit deux fichiers : *Animal.class* et *Coordonnées.class*.

?? **Les classes imbriquées ou *static***

Il peut être avantageux dans certains cas placer la définition d'une classe à l'intérieur d'une autre, lorsque celle-ci concerne uniquement « la classe principale ». Voyons pour notre exemple :

```
Class Animal {

   // champs

   boolean vivant ;
   int âge ;
   Coordonnées position ;

   // classes imbriquées

   static Class Coordonnées {

         // champs

         int x = 0;
         int y = 0;

         ...

      }

   // constructeurs

   Animal() {
         position = new Coordonnées() ;
         }

   ...

   }
```

La définition de la classe *Coordonnées* est maintenant imbriquée dans la classe *Animal*. Par ailleurs, la référence à la classe *Coordonnées* devient *Animal.Coordonnées*. De manière générale, les références à une classe imbriquée en Java se font en utilisant le point comme séparateur.

Lors de la compilation du fichier ci-dessus, *Animal.java*, le compilateur produit deux fichiers : *Animal.class* et *Animal$Coordonnées.class* pour la classe imbriquée.

Quant au chemin d'accès, notons que *mesclasses.Animal* désigne la classe *Animal* dans le package *mesclasse*, tandis que *mesclasses.Animal.Coordonnées* désigne la classe *Coordonnées* imbriquée dans la classe *Animal*, elle-même contenu dans le package *mesclassses*. Ainsi il est possible d'utiliser la directive *import* pour importer les classes imbriquées explicitement ou en bloc :

```
import mesclasses.Animal.Coordonnées ;
import mesclasses.Animal.* ;
```

Les classes imbriquées peuvent elles-mêmes contenir d'autres classes imbriquées, sans limitation de profondeur, du moins du point de vue de Java.

Un dernier point. La classe *Coordonnées* a été déclarée *static*, ce qui est obligatoire pour toute classe imbriquée. En revanche, les interfaces imbriquées sont automatiquement déclarées *static* et il n'est donc pas nécessaire de les déclarer explicitement comme telles.

?? **Les classes membres**

La particularité des classes imbriquées est qu'elles sont…

?? **Les classes locales**

?? **Les classes anonymes**

Les champs

Définition

L'état représente l'ensemble des variables qui caractérisent une classe ; on parle encore de champs ou de membres. Notons que Java initialise par défaut les variables membres.

Considérons l'exemple suivant :

```
Class Animal {

   // champs

   int âge ;
   static int longévité = 100 ;

}
```

Variables d'instances & Variables *static*

Dans l'exemple ci-dessus, *âge* est une variable d'instance, tandis que *longévité* représente une variable *static*.

Dans cet exemple, nous avons considéré que la *longévité* était une caractéristique commune à tous les animaux, mettons 100 ans ! Il n'est donc pas nécessaire de dupliquer cette information dans chacune des instances de la classe. Nous avons donc choisi de déclarer *longévité* comme une variable *static*. Il en résulte que cette variable appartient à la classe et non à ses instances.

Pour comprendre cette nuance, considérons une instance de la classe *Animal*, appelé *monAnimal*. L'objet *monAnimal* possède sa propre variable *âge*, à laquelle il est possible d'accéder grâce à la syntaxe :

```
monAnimal.âge
```

L'objet *monAnimal* ne possède pas de variable *longévité*. Normalement, *longévité* appartient à la classe *Animal*, et il est possible d'y accéder en utilisant la syntaxe :

```
Animal.longévité
```

Cependant, Java nous permet également d'utiliser la syntaxe :

```
monAnimal.longévité
```

Mais il faut bien comprendre que ces deux expressions font référence à la même variable. On peut utiliser le nom de la variable seul pour y faire référence, uniquement dans la définition de la classe.

Les variables *final*

Une variable déclarée *final* ne peut plus voir sa valeur modifiée. Elle remplit alors le rôle de constante dans d'autres langages. Une variable *final* est le plus souvent utilisée pour encoder des valeurs constantes.

Par exemple, on peut définir la constante *Pi* de la manière suivante :

```
final float pi = 3.14 ;
```

Déclarer une variable *final* offre deux avantages. Le premier concerne la sécurité. En effet, le compilateur refusera toute affectation ultérieure d'une valeur à la variable. Le deuxième avantage concerne l'optimisation du programme. Sachant que la valeur en question ne sera jamais modifié, le compilateur est à même de produire un code plus efficace. En outre, certains calculs préliminaires peuvent être effectués.

Les méthodes

Les méthodes sont les opérations ou les fonctions que l'on peut effectuer sur une classe. On distingue deux types de méthodes :

- les *accesseurs*, qui ne modifient pas l'état et se contente de retourner la valeur d'un champs ;
- les *modificateurs*, qui modifient l'état en effectuant un calcul spécifique.

Une déclaration de méthode est de la forme suivante :

```
[Modificateurs] Type nomMéthode ( paramètres ...)

      {
            corps de la méthode
      }
```

Le nom de la méthode débute par une minuscule ; la coutume veut qu'un accesseur débute par le mot « *get* » et qu'un modificateur débute par le mot « *set* ».

Les retours

Les méthodes d'instances

Les méthodes *static*

Les méthodes peuvent également être déclaré *static*. Imaginons que nous souhaitons construire un *accesseur* pour la variable *longévité* (voir exemple précédent). Nous pouvons le faire de la façon suivante :

```
Class Animal {

   // champs

   int âge ;
   static int longévité = 100 ;

   // méthodes

   static int getLongévité() {
         return longévité ;
   }

}
```

La méthode *getLongévité* peut être déclaré *static* car elle ne fait référence qu'à des membres *static* (en l'occurrence, la variable *longévité*). Ce n'est pas une obligation. Le programme fonctionne aussi si la méthode n'est pas déclaré *static*. Dans ce cas, cependant, la méthode est dupliquée chaque fois qu'une instance est créée, ce qui n'est pas très efficace.

Comme dans le cas des variables, les méthodes *static* peuvent être référencées à l'aide du nom de la classe ou du nom de l'instance. On peut utiliser le nom de la méthode seul , uniquement dans la définition de la classe.

Il est important de noter que les méthodes *static* ne peuvent en aucun cas faire référence aux méthodes ou aux variables non *static* de la classe. Elles ne peuvent non plus faire référence à une instance. (La référence *this* ne peut pas être employé dans la méthode *static*.)

Les méthodes *static* ne peuvent pas non plus être redéfinies dans les classes dérivées.

Les méthodes *native*

Une méthode peut également être déclarée *native*, ce qui a des conséquences importantes sur la façon de l'utiliser. En effet, une méthode *native* n'est pas écrite en Java, mais dans un autre langage. Les méthodes *native* ne sont donc pas portable d'un environnement à un autre. Les méthodes *native* n'ont pas de définition. Leur déclaration doit être suivie d'un point-virgule. (…)

Les méthodes *final*

Les méthodes peuvent également être déclaré *final*, ce qui restreint leur accès d'une toute autre façon. En effet, les méthodes *final* ne peuvent pas être redéfinies dans les classes dérivées. Ce mot clé est utilisé pour s'assurer que la méthode d'instance aura bien le fonctionnement déterminé dans la classe parente. (S'il s'agit d'une méthode *static*, il n'est pas nécessaire de la déclarer *final* car les méthodes *static* ne peuvent jamais être redéfinies.)

Les méthodes *final* permettent également au compilateur d'effectuer certaines optimisations qui accélèrent l'exécution du code. Pour déclarer une méthode *final*, il suffit de placer ce mot clé dans sa déclaration de la façon suivante :

```
final int calcul(int i, int j) {...}
```

Le fait que la méthode soit déclaré *final* n'a rien à voir avec le fait que ces arguments le soient ou non.

Nous reviendrons sur l'utilité des méthodes *final* dans le chapitre concernant le *polymorphisme*, et notamment le concept *early & late binding*.

Les constructeurs

Nous allons maintenant nous intéresser au début de la vie des objets : leur création et leur initialisation. Puis nous dirons aussi un mot sur la fin de vie des objet en traitant du *garbage collector.*

Les constructeurs : création d'objets

Les constructeurs et les initialiseurs sont des éléments très importants car ils déterminent la façon dont les objets Java commencent leur existence. Ces mécanismes, servant à contrôler la création d'instance de classe, sont fondamentalement différents des méthodes.

?? **Les constructeurs (*constructor*)**

Les constructeurs sont des méthodes particulières en ce qu'elles portent le même nom que la classe à laquelle elles appartiennent. Elles sont automatiquement exécutées lors de la création d'un objet. Le constructeur par défaut ne possèdent pas d'arguments.

- Les constructeurs n'ont pas de type et ne retournent pas.

- Les constructeurs ne sont pas hérités par les classes dérivées.

- Lorsqu'un constructeur est exécuté, les constructeurs des classes parentes le sont également. C'est *le chaînage des constructeurs*. Plus précisément, si en première instruction le compilateur ne trouvent pas un appel à *this(…)* ou *super(…)*, il rajoute un appel à *super(…)*. L'utilisation de *this(…)* permet de partager du code entre les constructeurs d'une même classe, dont l'un au moins devra faire référence au constructeur de la super-classe.

- Une méthode peut porter le même nom qu'un constructeur, ce qui est toutefois formellement déconseillé.

?? **Exemple de constructeurs**

Considérons l'exemple suivant :

```
class Animal {

   // champs

   boolean vivant ;
   int âge ;

   // constructeurs

   Animal() {
   }

   Animal(int a) {
         âge = a ;
         vivant = true ;
   }

   // méthodes

}
```

Si nous avions donné au paramètre *a* le même nom que celui du champs *âge*, il aurait fallu accéder à celle-ci de la façon suivante :

```
   Animal(int âge) {
         this.âge = âge ;
         vivant = true ;
   }
```

Toutefois, pour plus de clarté, il vaut mieux leur donner des noms différents. Dans le cas de l'initialisation d'une variable d'instance à l'aide d'un paramètre, on utilise souvent pour le nom du paramètre la première (ou les premières) lettre(s) du nom de la variable d'instance.

Par ailleurs, il faut remarqué que l'accès à un paramètre à l'intérieur d'une méthode est toujours plus rapide que l'accès à une variable d'instance. Le gain de performance est de 20%. Par conséquent, il faut utiliser les paramètres chaque fois que c'est possible.

?? **Création d'objets (*object*)**

Tout objet *java* est une *instance* d'une classe. Pour allouer la mémoire nécessaire à cet objet, on utilise l'opérateur *new*, qui lance l'exécution du constructeur.

La création d'un *Animal* se fait à l'aide de l'instruction suivante :

```
   Animal nouvelAnimal = new Animal(3) ;
```

?? **Surcharger les constructeurs**

Les constructeurs, tout comme les méthodes, peuvent être surchargés dans le sens où il peut y avoir plusieurs constructeurs dans une même classe, qui possèdent le même nom (celui de la classe). Un constructeur s'identifie de part sa signature qui doit être différente d'avec tous les autres constructeurs.

Supposons que la plupart des instances soient créées avec 0 pour valeur initiale de *âge*. Nous pouvons alors réécrire la classe *Animal* de la façon suivante :

```
class Animal {

   // champs

   boolean vivant ;
   int âge ;

   // constructeurs

   Animal() {
         âge = 0 ;
         vivant = true ;
   }

   Animal(int a) {
         âge = a ;
         vivant = true ;
   }

   // méthodes

}
```

Ici, les deux constructeurs possèdent des signatures différentes. Le constructeur sans paramètre traite le cas où l'âge vaut 0 à la création de l'instance. Une nouvelle instance peut donc être créée sans indiquer l'âge de la façon suivante :

```
Animal nouvelAnimal = new Animal() ;
```

?? **Autorisation d'accès aux constructeurs**

Les constructeurs peuvent également être affectés d'une autorisation d'accès. Un usage fréquent de cette possibilité consiste comme pour les variables, à contrôler leur utilisation, par exemple pour soumettre l'instanciation à certaines conditions.

Initialisation des objets

Il existe en Java trois éléments pouvant servir à l'initialisation :
- *les constructeurs*,
- *les initialiseurs de variables d'instances et statiques*,
- *les initialiseurs d'instances et statiques*.

Nous avons déjà présenté l'initialisation utilisant un constructeur. Voyons les deux autres manières.

?? **Les initialiseurs de variables d'instances et statiques**

Considérons la déclaration de variable suivante :

```
int a ;
```

Si cette déclaration se trouve dans une méthode, la variable n'a pas de valeurs. Toute tentative d'y faire référence poduit une erreur de compilation.

En revanche, s'il s'agit d'une *variable d'instance* (dont la déclaration se trouve en dehors de toute méthode), Java l'initialise automatiquement au moment de l'instanciation avec une valeur par défaut. Pour *les variables statiques*, l'initialisation est réalisée une fois pour toute à la première utilisation de la classe.

Les variables de type numérique sont initialisées à *0*. Le type booléen est initialisé à *false*.

Nous pouvons cependant initialiser nous-mêmes les variables de la façon suivante :

```
int a = 1 ;
int b = a*7 ;
float c = (b–c)/3 ;
boolean d = (a < b) ;
```

Les initialiseurs de variables permettent d'effectuer des opérations d'une certaine complexité, mais celle-ci est tout de même limitée. En effet, ils doivent tenir sur une seule ligne. Pour effectuer des opérations plus complexes, il convient d'utiliser les constructeurs ou encore les initialiseurs d'instances.

?? **Les initialiseurs d'instances**

Un initialiseur d'instance est tout simplement placé, comme les variables d'instances, à l'extérieur de toute méthode ou constructeur.

Voyons l'exemple suivant :

```
class Exemple {

   // champs

   int a ;
   int b ;
   float c ;
   boolean d ;

   // initialiseurs

   {
         a = 1 ;
         b = a*7 ;
         c = (b-a)/3 ;
         d = (a < b);
   }

}
```

Les initialisateurs d'instances permettent de réaliser des initialisations plus complexes. Ils offrent en outre l'avantage par rapport aux constructeurs d'être beaucoup plus rapides. De plus les initialisateurs permettent d'utiliser les exceptions pour traiter les conditions d'erreurs. Un autre avantage des initialisateurs est qu'ils permettent d'effectuer un traitement quelle que soit la signature du constructeur appelé. Il est ainsi possible de placer le code d'initialisation commun à tous les constructeurs dans un initialiseur et de ne traiter dans les différents constructeurs que les opérations spécifiques.

Les initialisateurs comportent cependant des limitations. Il n'est pas possible de leur passer des paramètres comme dans le cas des constructeurs. De plus, ils sont exécutés avant les constructeurs et ne peuvent donc utiliser les paramètres de ceux-ci.

?? **Les initialiseurs statiques**

Un *initialiseur statique* est semblable à un *initialiseur d'instance*, mais il est précédé du mot *static*. Considérons l'exemple suivant :

```
class Voiture {

   // champs

   static int capacité ;
```

```
	// initialiseurs

	static {
		capacité = 80;
		System.out.println("La variable vient d'être initialisée.\n") ;
	}

	// constructeurs

	Voiture() {
	}

	// méthodes

	static int getCapacité() {
		return capacité;
		}

}
```

L'initialiseur statique est exécuté au premier chargement de la classe, que ce soit pour utiliser un membre statique, `Voiture.getCapacité()` ou pour l'instancier, `Voiture maVoiture = new Voiture()`.

Les membres statiques (ici la variable `capacité`) doivent être déclarés avant l'initialiseur. Il est possible de placer plusieurs initialiseurs statiques, où l'on souhaite dans la classe. Ils seront tous exécutés au premier chargement de celle-ci, dans l'ordre où ils apparaissent.

?? **Les variables *final* non initialisées**

Les finaliseurs

La destruction des objets (*garbage collector*)

Avec certains langages, le programmeur doit s'occuper lui-même de libérer la mémoire en supprimant les objets devenus inutiles. Avec Java, le problème est résolu de façon très simple : un programme, appelé *garbage collector*, ce qui signifie littéralement « ramasseur d'ordures », est exécuté automatiquement dès que la mémoire disponible devient inférieure à un certain seuil. De cette façon, aucun objet inutilisé n'encombrera la mémoire.

Le concept de l'héritage

Hiérarchie des classes

De plus chaque classe *dérive* d'une classe de niveau supérieur, appelée *sur-classe*. Cela est vrai pour toutes les classes sauf une. Il s'agit de la classe *Object*, qui est l'ancêtre de toutes les classes.

Toute instance d'une classe est un objet du type correspondant, mais aussi du type de toutes ses classes ancêtres.

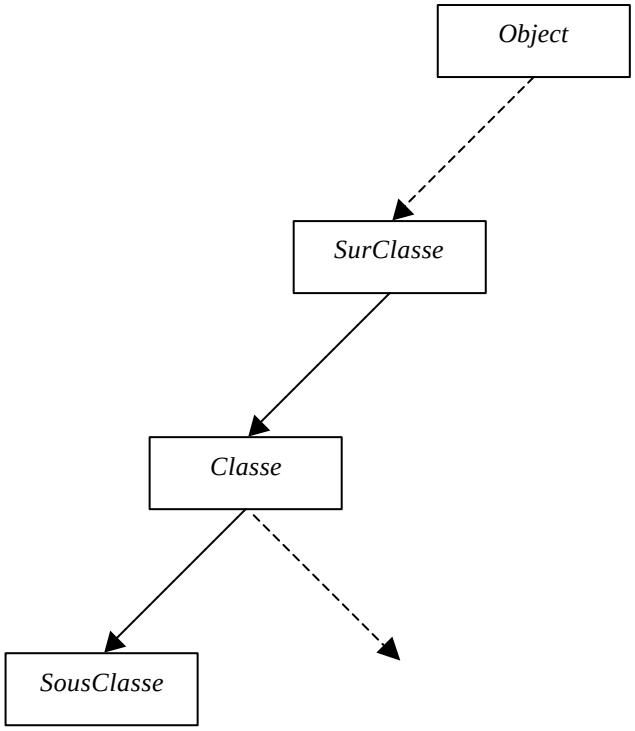
SousClasse hérite de toutes les caractéristiques de *Classe*, et donc, par transitivité, de *SurClasse*.

Une *classe* est toujours construite à partir d'une autre classe dont elle est *dérivée*. Une classe dérivée est une *sous-classe* d'une *sur-classe*.

?? **Extends**

Lorsque le paramètre *extends* est omis, la classe déclarée est une sous classe de l'objet *Objet*.

?? **Référence à la classe parente**



Redéfinition des champs et des méthodes

?? **Redéfinition des méthodes**

Une deuxième déclaration d'une méthode dans une classe dérivée remplace la première. Voyons un exemple avec la méthode *crie( )* redéfinie dans la classe *Chien* dérivée de la classe *Animal*.

```
Class Animal {

   // champs

   // méthodes

   void crie() {
   }

}

Class Chien extends Animal {

   // champs

   // méthodes

   void crie() {
         System.out.println("Ouah-Ouah !") ;
   }

}
```

La surchage

?? **Surcharger les méthodes**

Signature d'une méthode…
Une méthode est dite surchargée si elle permet plusieurs passages de paramètres différents.

Accessibilité

En Java, il existe quatre catégories d'autorisations d'accès, spécifiés par les modificateurs suivants : *private, protected, public*. La quatrième catégorie correspond à l'absence de modificateur.

Nous allons les présenter en partant du moins restrictif jusqu'au plus restrictif.

?? **public**

Les classes, les interfaces, les variables (primitives ou objets) et les méthodes peuvent être déclarées *public*.

Les éléments *public* peuvent être utilisés par n'importe qui sans restriction ; il est accessible à l'extérieur de la classe.

?? **protected**

Cette autorisation s'applique uniquement aux membres de classes, c'est-à-dire aux variables (objets ou primitives), aux méthodes et aux classes internes. Les classes qui ne sont pas membre d'une autre classe ne peuvent pas être déclarées *protected*.

Dans ce cas, l'accès en est réservé aux méthodes des classes appartenant au même *package*, aux classes dérivées de ces classes, ainsi qu'aux classes appartenant aux mêmes *packages* que les classes dérivées. Plus simplement, on retiendra qu'un élément déclaré *protected* n'est visible que dans la classe où il est défini et dans ses sous-classes.

?? **Autorisation par défaut**

L'autorisation par défaut s'applique aux classes, interfaces, variables et méthodes.

Les éléments qui disposent de cette autorisation sont accessibles à toute les méthodes des classes du même package. Les classes dérivées ne peuvent donc y accéder que si elles sont explicitement déclarées dans le même package.

Rappelons que les classes n'appartenant pas explicitement à un package appartiennent automatiquement au package par défaut. Toute classe sans indication de package dispose donc de l'autorisation d'accès à toutes les classes se trouvant dans le même cas.

?? **private**

L'autorisation *private* est la plus restrictive. Elle s'applique aux membres d'une classe (variables, méthodes, classes internes).

Les éléments déclarés *private* ne sont accessibles que depuis la classe qui les contient ; il n'est visible que dans la classe où il est défini.

Ce type d'autorisation est souvent employé pour les variables qui ne doivent être lues ou modifiées qu'à l'aide d'un *accesseur* ou d'un *modificateur*. Les accesseurs et les modificateurs, de leur côté, sont déclarés *public*, afin que tout le monde puisse utiliser la classe.

Les classes abstraites, les interfaces, le polymorphisme

Le mot clé *abstract*

?? **Méthodes et classes abstraites**

Une méthode déclarée *abstract* ne peut être exécutée. En fait, elle n'a pas d'existence réelle. Sa déclaration indique simplement que les classes dérivées doivent la redéfinir.

Les méthodes *abstract* présentent les particularités suivantes :

- Une classe qui contient une méthode *abstract* doit être déclarée *abstract*.
- Une classe *abstract* ne peut pas être instanciée.
- Une classe peut être déclarée *abstract*, même si elle ne comporte pas de méthodes *abstract*.
- Pour pouvoir être instanciée, une sous-classe d'une classe *abstract* doit redéfinir toute les méthodes *abstract* de la classe parente.
- Si une des méthodes n'est pas redéfinie de façon concrète, la sous-classe est elle-même *abstract* et doit être déclarée explicitement comme telle.
- Les méthodes *abstract* n'ont pas d'implémentation. Leur déclaration doit être suivie d'un point-virgule.

Ainsi dans l'exemple précédent la méthode *crie( )* de la classe *Animal* aurait pu être déclarée *abstract*, ce qui signifie que tout *Animal* doit être capable de crier, mais que le cri d'un animal est une notion abstraite. La méthode ainsi définie indique qu'une sous-classe devra définir la méthode de façon concrète.

```
abstract class Animal {

   // champs

   // méthodes

   abstract void crie() ;
}

class Chien extends Animal {

   // champs

   // méthodes

   void crie() {
         System.out.println("Ouah-Ouah !") ;
   }

}

class Chat extends Animal {

   // champs

   // méthodes

   void crie() {
         System.out.println("Miaou-Miaou !") ;
   }
```

```
}
```

De cette façon, il n'est plus possible de créer un animal en instanciant la classe *Animal*. En revanche, grâce à la déclaration de la méthode *abstract crie( )* dans la classe *Animal*, il est possible de faire crier un animal sans savoir s'il s'agit d'un chien ou d'un chat, en considérant les instances de Chien ou de Chat comme des instances de la classe parente.

```
Animal animal1 = new Chien() ;
Animal animal2 = new Chat() ;

animal0.crie() ;
animal1.crie() ;
```

Le premier animal crie "Ouah-Ouah !" ; le second, "Miaou-Miaou !".

Les interfaces

Une classe peut contenir des méthodes *abstract* et des méthodes non *abstract*. Cependant, il existe une catégorie particulière de classes qui ne contient que des méthodes *abstract*. Il s'agit des interfaces. Les interfaces sont toujours *abstract*, sans qu'il soit nécessaire de l'indiquer explicitement. De la même façon, il n'est pas nécessaire de déclarer leurs méthodes *abstract*.

Les interfaces obéissent par ailleurs à certaines règles supplémentaires.

- Elles ne peuvent contenir que des variables *static* et *final*.

- Elles peuvent être étendues comme les autres classes, avec une différence majeure : une interface peut dériver de plusieurs autres interfaces. En revanche, une classe ne peut pas dériver uniquement d'une ou de plusieurs interfaces. Une classe dérive toujours d'une autre classe, et peut dériver, en plus, d'une ou plusieurs interfaces.

-

Casting

?? **Sur-casting**

Un objet peut être considéré comme appartenant à sa classe ou à une classe parente selon le besoin, et cela de façon dynamique. Nous rappelons ici que toutes classes dérive de la classe *Object*, qui est un type commun à tous les objets. En d'autres termes, le lien entre une classe et une instance n'est pas unique et statique. Au contraire, il est établi de façon dynamique, au moment où l'objet est utilisé. C'est la première manifestation du polymorphisme !

Le sur-casting est effectué de façon automatique par Java lorsque cela est nécessaire. On dit qu'il est implicite. On peut l'expliciter pour plus de clarté, en utilisant l'opérateur de casting :

```
Chien chien = new Chien() ;
Animal animal = (Animal)chien ;
```

Après cette opération, ni le handle *chien*, ni l'objet correspondant ne sont modifiés. Les handles ne peuvent jamais être redéfinie dans le courant de leur existence. Seule la nature du lien qui lie l'objet aux handles change en fonction de la nature des handles.

Le sur-casting est un peu moins explicite, lorsqu'on affecte un objet à un handle de type différent. Par exemple :

```
Animal animal = new Chien() ;
```

?? **Sous-casting**

Le sous-casting doit obligatoirement être explicite en Java.

Polymorphisme

?? **Utilisation du sur-casting**

Considérons l'exemple suivant, qui reprend les définitions précédentes des classes *Animal*, *Chien*, et *Chat* et illustre une première sorte de polymorphisme avec sur-casting implicite sur la méthode *crie( )*.

```
public class Main {

   // méthodes

   public static void main(String[] argv) {
         Chien chien = new Chien() ;
         Chat chat = new Chat() ;
         crie(chien) ;
         crie(chat) ;

   void crie(Animal animal) {
         animal.crie() ;
   }
}
```

Le premier animal crie "Ouah-Ouah !" ; le second, "Miaou-Miaou !".

?? ***Late-binding***

Il existe un moyen d'éviter le sous-casting explicite en Java, appelé *late-binding*. Cette technique fondamentale du polymorphisme permet de déterminer dynamiquement quelle méthode doit être appelée.

Dans la plupart des langages, lorsque le compilateur rencontre un appel de méthode, il doit être à même de savoir exactement de quelle méthode il s'agit. Le lien entre l'appel et la méthode est alors établi à la compilation. Cette technique est appelée *early binding* (liaison précoce). Java utilise cette technique pour les appels de méthodes déclarées *final*. Elle a l'avantage de permettre certaines optimisations.

En revanche, pour les méthodes qui ne sont pas *final*, Java utilise la technique du *late binding* (liaison tardive). Dans ce cas, le compilateur n'établit le lien entre l'appel et la méthode qu'au moment de l'exécution du programme. Ce lien est établi avec la version la plus spécifique de la méthode et doit être différencié du concept *abstract*. Considérons l'exemple suivant pour s'en convaincre.

```
class Animal {

   // méthodes

   void crie() {
   }
}

class Chien extends Animal {

   // méthodes

   void crie() {
         System.out.println("Ouah-Ouah !") ;
   }

}
```

```
   class Chat extends Animal {

      // méthodes

      void crie() {
            System.out.println("Miaou-Miaou !") ;
      }
   }

   public class Main {

      // méthodes

      public static void main(String[] argv) {
            Chien chien = new Chien() ;
            Chat chat = new Chat() ;
            crie(chien) ;
            crie(chat) ;

      void crie(Animal animal) {
            animal.crie() ;
      }
}
```

Le premier animal crie "Ouah-Ouah !" ; le second, "Miaou-Miaou !". La méthode *crie* appelé dans la méthode *crie* de la classe *Main* est bien la plus spécifique, celle de *Chien* ou de *Chat* et non celle de *Animal* !

?? ***Polymorphisme***

Le programme ci-dessous illustre le concept du polymorphisme. La classe *Animal* utilise la méthode abstraite *qui* pour définir la méthode *printQui* de manière plus ou moins abstraite. Cela entraîne une factorisation du code.

```
   abstract class Animal {

      // méthodes

      abstract String qui() ;

      void printQui() {
            System.out.println("cet animal est un " + qui()) ;
      }
   }

   class Chien extends Animal {

      // méthodes

      String qui() {
            return "chien" ;
      }
   }

   class Chat extends Animal {

      // méthodes

      String qui() {
            return "chat" ;
      }
```

```
}
```

L'implémentation de la méthode *qui* est relié à l'appel, uniquement au moment de l'exécution, en fonction du type de l'objet appelant et non celui du handle !
C'est-à-dire,

```
Animal animal1 = new Chien() ;
Animal animal2 = new Chat() ;
animal1.printQui() ;
animal2.printQui() ;
```

donne comme résultat :

```
cet animal est un chien
cet animal est un chat
```

Chapitre 3 – Spécificités du langage

Les entrées / sorties

Package

Les packages

?? **Les packages accessibles par défaut**

?? **L'instruction *package***

?? **L'instruction *import***

Le clonage

Le clonage est une technique de création d'objets…

Les threads

Les *threads* (en français processus indépendants) sont des mécanismes importants du langage Java. Ils permettent d'exécuter plusieurs programmes indépendants les uns des autres. Ceci permet une exécution parallèle de différentes tâches de façon autonome.

Un *thread* réagit aux différentes méthodes suivantes :

- *destroy( )* : arrêt brutal du *thread* ;
- *interrupt( )* permet d'interrompre les différentes méthodes d'attente en appelant une exception ;
- *sleep( )* met en veille de *thread* ;
- *stop( )* : arrêt non brutal du *thread* ;
- *suspend( )* : arrêt d'un *thread* en se gardant la possibilité de le redémarrer par la méthode *resume( ) ;*
- *wait( )* met le *thread* en attente ;
- *yield( )* donne le contrôle au schéduleur.

La méthode *sleep( )* est souvent employée dans les animations, elle permet de mettre des temporisations d'attente entre deux séquences d'image par exemple.

Programme principal : la méthode *main*

(…)

Cette méthode doit impérativement être déclaré public. Ainsi la classe contenant la méthode *main( )*, le programme principal, doit être *public*. Rappelons ici qu'un fichier contenant un programme Java ne peut contenir qu'une seule définition de classe déclarée *public*. De plus le fichier doit porter le même nom que la classe, avec l'extension *.java*.

Les exceptions (*exception*) et les erreurs (*error*)

?? Deux types d'erreurs en Java

En Java, on peut classer les erreurs en deux catégories :
- *les erreurs surveillées*,
- *les erreurs non surveillées*.

Java oblige le programmeur à traiter les erreurs surveillées. Les erreurs non surveillées sont celles qui sont trop graves pour que le traitement soit prévu à priori., comme par exemple la division par zéro.

```
Public class Erreur {
      Public static void main (String[] args) {
            int x = 10, y = 0, z = 0;
            z = x / y ;
      }
}
```

Dans ce cas, l'interpréteur effectue *un traitement exceptionnel*, il arrête le programme et affiche un message :

```
Exception in thread "main"
java.lang.ArithmeticException : / by zero
      at Erreur.main(Erreur.java:5)
```

?? Principe

Lorsqu'une erreur de ce type est rencontrée, l'interpréteur crée immédiatement un objet, instance d'une classe pariculière, elle-même sous-classe de la classe ***Exception***. Cet objet est créé normalement avec l'opérateur *new*. Puis l'interpréteur part à la recherche d'une portion de code capable de recevoir cet objet et d'effectuer le traitement approrié.

S'il s'agit d'une *erreur surveillée* par le compilateur, celui-ci a obligé le programmeur à fournir ce code. Dans le cas contraire, le traitement est fourni par l'interpréteur lui-même. Cette opération est appelée *lancement d'une exception* (***throw***). Pour trouver le code capable de traiter l'objet, l'interpréteur se base sur le type de l'objet, c'est-à-dire sur la classe dont il est une instance.

Pour reprendre l'exemple de la division par zéro, une instance de la classe `ArithmeticException` est lancée. Si le programme ne comporte aucun bloc de code capable de traiter cet objet, celui-ci est attrapé par l'interpréyeur lui-même. Un message d'erreur est alors affichée `Exception in thread "main".`

?? Attraper les exceptions

Nous avons vu que Java n'oblige pas le programmeur à attrapper tous les types d'exceptions. Seuls ceux correspondant à des *erreurs surveillées* doivent obligatoirement être attrapés. En fait, les exceptions qui ne peuvent pas être attrapées sont des instances de la classe ***RuntimeException*** ou une classe dérivée de celle-ci.

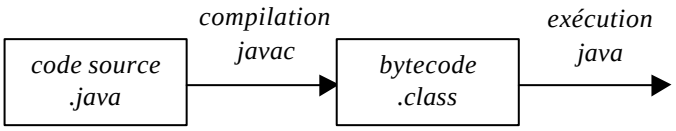
Cependant rien n'interdit d'attraper ces exceptions. Nous avons dit…

Annexes

La machine virtuelle Java (*JVM*)

Java est un langage *multi-plates-formes* qui permet, selon *Sun Microsystems*, son concepteur, d'écrire une fois pour toute des applications capables de fonctionner dans tous les environnements. Cet objectif est atteint grâce à l'utilisation d'une machine virtuelle Java (JVM) qui exécute les programmes écrits dans ce langage.

Compilation



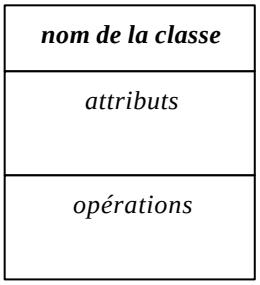
- compilation du code source : *javac *.java* ;
- exécution sur la JVM : *java MainFile*.

Diagramme de classe – UML

UML (Unified Modeling Language) propose une modélisation des langages unifiées. Nous allons l'utiliser particulièrement pour construire *un diagramme de classe*, qui modélise l'architecture des classes dans un programme *Java*.

Ce diagramme est construit à partir des classes créés par le programmeur (mises en gras) et il indique *les relations de dépendances* entre classes par des flèches, selon une typologie donnée.

Représentation d'une classe



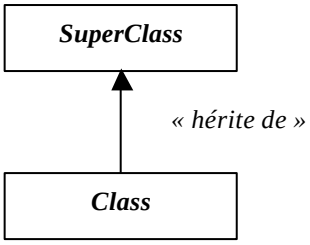
?? **visibilité**

+	*public*
#	*protégé*
-	*privé*

Relations de dépendances

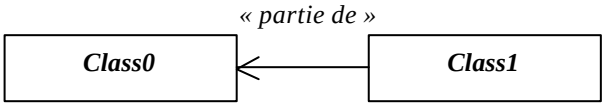
En UML, on distingue plusieurs types de relations de dépendances : *la généralisation, l'association, la généralisation particulière*.

?? **Généralisation – Relation d'héritage**

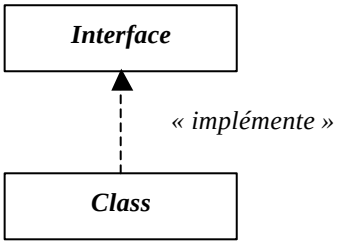


?? **Association – Relation de contenance**

Une classe contient une instance d'une autre classe.



?? **Généralisation particulière – Implémentation d'une interface**

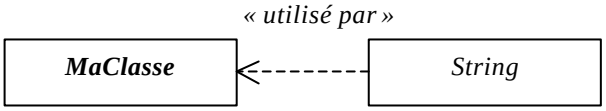


?? **Autres relations de dépendance**

Toutes les classes ne sont pas liés par une association ou une généralisation (par exemple : *String*…) ; il y a aussi des interactions entre les objets qui surviennent à l'exécution…

On peut les faire apparaître dans le diagramme de classe sous forme d'une relation de dépendance « *dependancy* » en précisant « *paramètre* » ou « *variable locale* ».

En fait, la relation de dépendance est très générale, elle indique qu'une classe a besoin de l'interface d'une autre classe ; on dit aussi qu'une classe est cliente d'une autre classe. Dans l'exemple ci-dessous, Mac*l*asse utilise des *String* comme paramètres de méthodes.



Remarquons que l'association et la généralisation sont aussi des relations de dépendances.

Diagramme de séquence – UML