Programmation orientée Objet

Dr Papa Samour DIOP

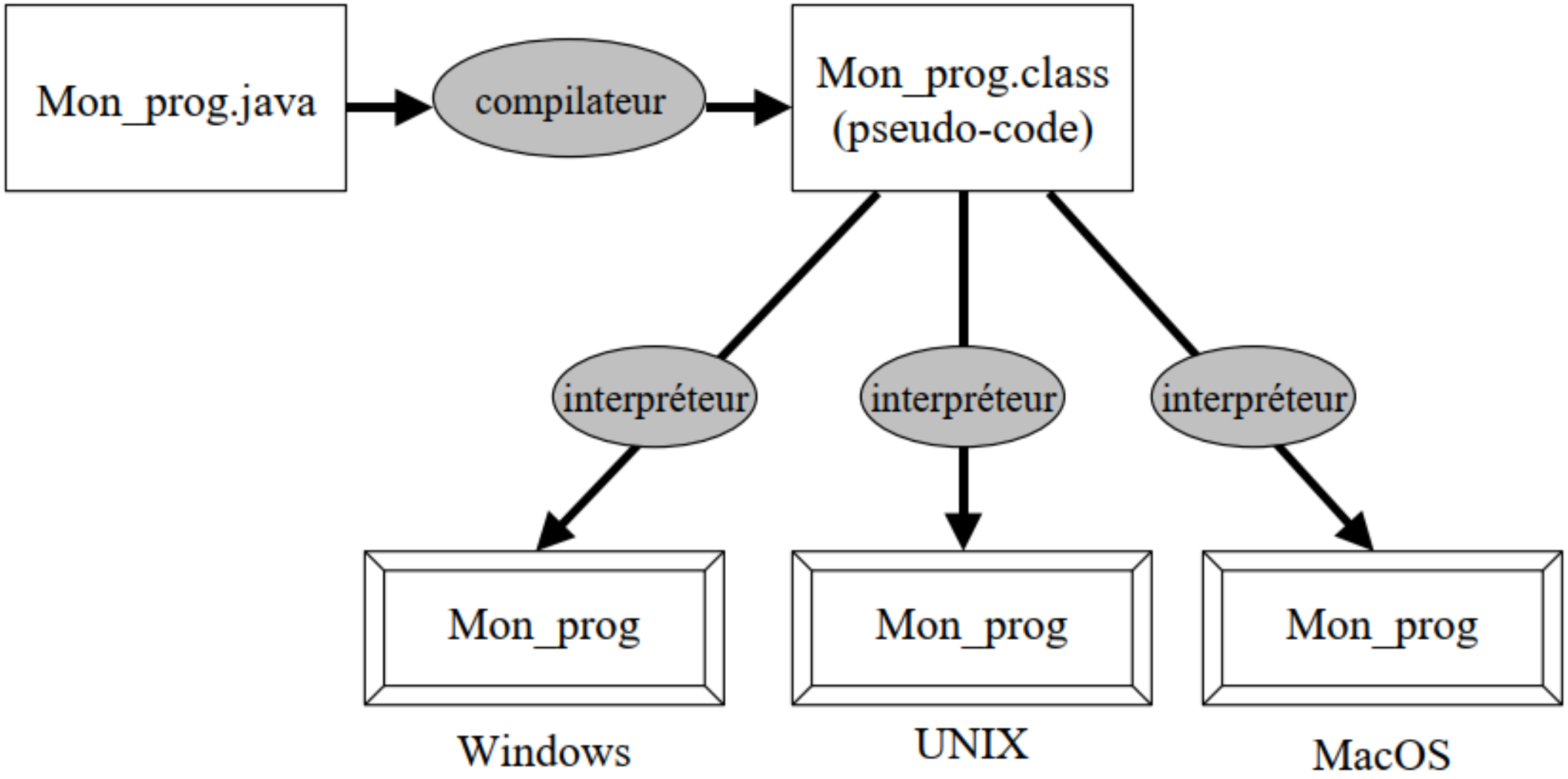
Enseignant chercheur IPSL/UGB

L’approche orientée objet (Début des années 80).

- Selon cette approche, un programme est vu comme un ensemble d’entités (ou objets). Au cours de son exécution, ces entités collaborent en s’envoient des messages dans un but commun

Java et la portabilité

- Dans la plupart des langages de programmation, un programme **portable** est un programme dont le code peut être exploité dans différents environnements moyennant simplement une nouvelle compilation.
- En Java, la portabilité va plus loin…
  En java, un programme est à la fois compilé et interprété.
- La compilation d'un programme Java produit un code intermédiaire exécutable formé de **bytecodes**, qui est un pseudo code universel disposant des fonctionnalités communes à toutes les machines.
- L'interpréteur Java appelé **machine virtuelle** (JVM pour Java Virtual Machine) permet d'exécuter le bytecode produit par le compilateur.


Mon_prog.java
compilateur
Mon_prog.class
(pseudo-code)
interpréteur
interpréteur
interpréteur
Mon_prog
Mon_prog
Mon_prog
Windows
UNIX
MacOS

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 ? 1.79769313486231 57E308	64 bits	Double
boolean	true ou false	1 bit	Boolean
void	0 bit	Void	

- Les classes *BigInteger* et *BigDecimal* sont utilisés pour représenter respectivement des valeurs entières et décimales de précision quelconque.
- 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.
- 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 !

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 !
- **final**
  - L'utilisation de final n'est pas réservé aux primitives. Un *handle* d'un objet peut parfaitement être déclaré *final*.

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.

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)

symbole	**description**	**arité**	**exemple**
opérateur d'affectation	=	affectation	x = 2
	–=	soustraction et affectation	x–= 2
	+=	addition et affectation	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	y – x
	*	multiplication	3 * x
	/	division	4 / 2
	%	modulo (reste de la division)	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 .
La *division* des entiers fournit un résultat tronqué et non arrondi.

Opérateurs à un opérande

symbole	descriptionarité		exemple
opérateurs à un opérande	–	opposé	–x
	++	pré-incrémentation	++x
	++	post-incrémentation	x++
	--	pré-décrémentation	--x
	--	post-décrémentation	x--

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

- 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 ;
.println(a == b) ;
.println(a == c) ;
.println(a.equals(b)) ;
```

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* (références).

Opérateurs logiques

symbole	**descriptionarité**		**exemple**
opérateurs logiques	&&	et	a && b
	||	ou	a || b
	!	non	!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

symbole	descriptionarité		exemple
opérateurs d’arithmétique binaire	&	et	a & b
	|	ou	a | b
	^	ou exclusif	a ^ b
	~	non	~a
	<<	décalage à gauche	a << 2
	>>	décalage à droite	b >> 2
	>>>	décalage à droite sans extension du signe	b >>> 2

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 ; ? Donner la valeur de a

- 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*.

Les structures de contrôle

if-else

L'expression if permet d'exécuter un bloc d'instructions uniquement si l'expression booléenne est évaluée à vrai :

```
if (i % 2 == 0) {
  // instructions à exécuter si i est pair
}
```

L'expression if peut être optionnellement suivie d'une expression else pour les cas où l'expression est évaluée à faux :

```
if (i % 2 == 0) {
  // instructions à exécuter si i est pair
} else {
  // instructions à exécuter si i est impair
}
```

- L’expression else peut être suivie d’une nouvelle instruction if afin de réaliser des choix multiples :

```
if (i % 2 == 0) {
  // instructions à exécuter si i pair
} else if (i > 10) {
  // instructions à exécuter si i est impair et supérieur à 10
} else {
  // instructions à exécuter dans tous les autres cas
}
```

Return

- return est un mot clé permettant d'arrêter immédiatement le traitement d'une méthode et de retourner la valeur de l'expression spécifiée après ce mot-clé. Si la méthode ne retourne pas de valeur (void), alors on utilise le mot-clé return seul. L'exécution d'un return entraîne la fin d'une structure de contrôle.

```
if (i % 2 == 0) {
  return 0;
}
```

- Écrire des instructions immédiatement après une instruction return n'a pas de sens puisqu'elles ne seront jamais exécutées. Le compilateur Java le signalera par une erreur unreachable code.

```
if (i % 2 == 0) {
  return 0;
  i++; // Erreur de compilation : unreachable code
}
```

While

- L’expression while permet de définir un bloc d’instructions à répéter tant que l’expression booléenne est évaluée à vrai.

```
while (i % 2 == 0) {
  // instructions à exécuter tant que i est pair
}
```

- L’expression booléenne est évaluée au départ et après chaque exécution du bloc d’instructions.

do-while

- Il existe une variante de la structure précédente, nommée do-while :

```
do {
  // instructions à exécuter
} while (i % 2 == 0);
```

- Dans ce cas, le bloc d'instruction est exécuté une fois puis l'expression booléenne est évaluée. Cela signifie qu'avec un do-while, le bloc d'instruction est exécuté au moins une fois.

for

- Une expression for permet de réaliser une itération. Elle commence par réaliser une initialisation puis évalue une expression booléenne. Tant que cette expression booléenne est évaluée à vrai, le bloc d'instructions est exécuté et un incrément est appelé.

```
for (initialisation; expression booléenne; incrément) {
  bloc d'instructions
}
```

```
for (int i = 0; i < 10; ++i) {
  // instructions
}
```

Exemple

```
public class PremProg
 {
public static void main(String args [ ] )
        {
          int i, total=0, n=5;


          for(i=1;i<=n;i++)
          total=total+i;
          System.out.println("somme des entiers de 1 a " +n+ " = " + total);
        }
}
```

break-continue

- Pour les expressions while, do-while, for permettant de réaliser des itérations, il est possible de contrôler le comportement à l'intérieur de la boucle grâce aux mots-clés break et continue.
- **break quitte la boucle sans exécuter le reste des instructions.**

```
int k = 10;
for (int i = 1 ; i < 10; ++i) {
  k *= i
  if (k > 200) {
    break;
  }
}
```

- **continue arrête l'exécution de l'itération actuelle et commence l'exécution de l'itération suivante.**

```
for (int i = 1 ; i < 10; ++i) {
  if (i % 2 == 0) {
    continue;
  }
  System.out.println(i);
}
```

switch

- Un expression switch permet d’effectuer une sélection parmi plusieurs valeurs.

```
switch (s) {
  case "valeur 1":
    // instructions
    break;
  case "valeur 2":
    // instructions
    break;
  case "valeur 3":
    // instructions
    break;
  default:
    // instructions
}
```

- switch évalue l’expression entre parenthèses et la compare dans l’ordre avec les valeurs des lignes case. Si une est identique alors il commence à exécuter la ligne d’instruction qui suit. Attention, un case représente un point à partir duquel l’exécution du code commencera. Si on veut isoler chaque cas, il faut utiliser une instruction break. Au contraire, l’omission de l’instruction break peut être pratique si on veut effectuer le même traitement pour un ensemble de cas :

```
switch (c) {
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
  case 'y':
    // instruction pour un voyelle
    break;
  default:
    // instructions pour une consonne
}
```

- On peut ajouter une cas default qui servira de point d'exécution si aucun case ne correspond.
- Par convention, on place souvent le cas default à la fin. Cependant, il agit plus comme un libellé indiquant la ligne à laquelle doit commencer l'exécution du code. Il peut donc être placé n'importe où :

```
switch (c) {
  default:
    // instructions pour une consonne
  case 'a':
  case 'e':
  case 'i':
  case 'o':
  case 'u':
  case 'y':
    // instructions pour les consonnes et les voyelles
}
```

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.

- **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. *abstract* est un modificateur de non-accès en Java applicable pour les classes, les méthodes mais **pas les** variables. Il est utilisé pour réaliser l'abstraction qui est l'un des piliers de la programmation orientée objet (POO).

- **private**
  - Le mot-clé **private** rend les variables d'instance et les méthodes private qui ne sont accessibles qu'à partir de la même classe.
- **public**
  - Le mot-clé **public** rend publiques les variables d'instance et les méthodes accessibles de l'extérieur de la classe.

- **Variables locales** - Les variables définies dans des méthodes, des constructeurs ou des blocs sont appelées variables locales. La variable sera déclarée et initialisée dans la méthode et sera détruite une fois la méthode terminée.
- **Variables d'instance** - Les variables d'instance sont des variables dans une classe mais en dehors de toute méthode. Ces variables sont initialisées lorsque la classe est instanciée. Les variables d'instance sont accessibles depuis n'importe quelle méthode, constructeur ou bloc de cette classe particulière.
- **Variables de classe** - Les variables de classe sont des variables déclarées dans une classe, en dehors de toute méthode, avec le mot clé **static**.

Concepts de base de la programmation orientée objet

- « 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.
  - 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 :

- 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 donc un objet de la classe Chien* .

Les classes

- 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() {
 }
                ++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.

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() ;
}
 }
```

- 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 , 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 age ;
 Coordonnées position ;
// classes imbriquées
static Class Coordonnees {
        // champs
     int x = 0;      int y = 0;
// constructeurs
Animal() { position = new Coordonnees() ;
}
 }
Class Coordonnees {
 // champs
int x = 0;  int y = 0;
 }
```

Les champs

- Notons que Java initialise par défaut les variables membres d'une classe.

Considérons l'exemple suivant :

```
Class Animal {
 // champs   int âge ;
 static int longevite = 100 ;
}
```

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

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 *accesseurs*, qui ne modifient pas l'état et se contente de retourner la valeur d'un champs (geteur et seteur) ;
    - 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 méthodes *static*

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

```
Class Animal {
 // champs
  int age ;
 static int longevite = 100 ;
 // méthodes
 static int getLongevite() {
                return longévite ;
 }
 }
```

- La méthode *getLongevite* peut être déclaré *static* car elle ne fait référence qu'à des membres *static* (en l'occurrence, la variable *longevite*). 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 *native***
  - Une méthode peut également être déclarée *native*, ce qui a des conséquences importantes sur la façon de l'utiliser. 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.

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 (*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.

Objet

- Les objets ont des états et des comportements. Exemple: Une Lampe a des isOn, ainsi que comportements - allumer, éteindre. Un objet est une instance d'une classe.

- **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 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 produit 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 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 capacite ;
// initialiseurs
static { capacite = 80;
.println("La variable vient d'être initialisée.\n") ;
            }
 // constructeurs
 Voiture() {
 }
 // méthodes
  static int getCapacite() {     return capacite;
             }
}
```

- L’initialiseur statique est exécuté au premier chargement de la classe, que ce soit pour utiliser un membre statique, **Voiture.getCapacite()** 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.

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.

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() {
                .println("Ouah-Ouah !") ;
 }
}
```

La surchage

Surcharger les méthodes

- 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.

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.
- 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 *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() {
                .println("Ouah-Ouah !") ;
 }
}
```

```
class Chat extends Animal {
 // champs
 // méthodes
 void crie() {
                .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() ;
- Le premier animal crie "Ouah-Ouah !" ; le second, "Miaou-Miaou !".
- 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.

Polymorphisme

Utilisation du sur-casting

```
 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) {
() ;
 }
}
```

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() {
                .println("Ouah-Ouah !") ;
 }
}
```

```
class Chat extends Animal {
 // méthodes
 void crie() {
                .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) ;
 }
}
```

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() {
                .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

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* ;
  - *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 ;
  - 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"
- .ArithmeticException : / by zero at (:5)

Principe

- 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.