http://www.labo-sun.com
labo-sun@supinfo.com

LABORATOIRE SUPINFO DES TECHNOLOGIES Sun

Les classes

CONCEPTS & HERITAGE

Auteur : Renaud Roustang
Version 2.0 – 14 octobre 2004
Nombre de pages : 24

Ecole Supérieure d'Informatique de Paris
23. rue Château Landon 75010 – PARIS
www.supinfo.com

Table des matières

1. POO PRESENTATION 5

1.1. NOTION D'OBJET 5
1.2. NOTION DE CLASSE 5
1.3. LE PRINCIPE D'ENCAPSULATION 5

2. LES CLASSES 6

2.1. CREATION D'UNE CLASSE 6
2.2. LE CONSTRUCTEUR 6
2.2.1. Définition 6
2.2.2. Surdéfinition du constructeur 7
2.2.3. Surcharge du constructeur 7
2.3. INSTANCIER UNE CLASSE 7
2.3.1. Définition 7
2.3.2. L'opérateur new 8
2.3.3. Affectation d'objets 8
2.4. LA METHODE MAIN 9
2.4.1. Définition 9
2.4.2. Contraintes 9
2.4.3. Paramètres 9
2.5. L'OPERATEUR : "." ET LE POINTEUR "THIS" 10
2.5.1. L'opérateur "." 10
2.5.2. Le pointeur "this" 10

3. UTILISATION DES PACKAGES 12

3.1. PACKAGE 12
3.1.1. Définition 12
3.1.2. Utilisation 12
3.2. IMPORT 13

4. HERITAGE 14

4.1. DEFINITION 14
4.2. EXTENDS 14
4.3. SUPER 15

5. LES INTERFACES (HERITAGE MULTIPLE) 16

5.1. DEFINITION 16
5.2. IMPLEMENTATION 16
5.3. COMPARAISON AVEC LES CLASSES ABSTRAITES 16

6. MODIFICATEURS DE VISIBILITE 18

6.1. PUBLIC 18
6.2. PRIVATE 18
6.3. PROTECTED 18
6.4. AUCUN MODIFICATEUR 19

7. AUTRES MODIFICATEURS 20

7.1. ABSTRACT 20
7.1.1. Définition 20
7.1.2. Utilisation 20
7.1.3. Intérêt 20
7.2. FINAL 21
7.2.1. Variable finale 22

7.2.2. Méthode finale *22*
7.2.3. Classe finale *22*
7.3. STATIC 22
8. GARBAGE COLLECTOR 24
8.1. ROLE 24
8.2. METHODE FINALIZE() 24
8.3. CONTROLE DU GARBAGE COLLECTOR 24

1. POO présentation

1.1. Notion d'objet

Les entités informatiques qui peuvent se définir en un ensemble d'entités sont appelées objet. Pour expliquer la programmation objet on peut faire une analogie avec l'homme qui classe tout ce qui l'entoure. La difficulté majeure consiste à modéliser les objets.

Les caractéristiques d'un objet :

Les méthodes : Actions que peut effectuer un objet.

Les attributs : Données stockant des informations sur un objet.

L'identité : Identifiant unique attribué a un objet lui permettant d'être distingué des autres objets.

1.2. Notion de classe

Les classes représentent la structure d'un objet (instance), qui peut être définie comme l'ensemble des éléments qui composent cet objet. L'objet découle d'une classe. Les méthodes (opérations appliquées aux objets) et les attributs (valeurs d'un objet) sont les deux types d'éléments qui constituent les classes.

Prenons un cas concret :

La race « terrier » représente la classe principale.
Bull terrier et fox terrier sont des instanciations de la classe terrier.

1.3. Le Principe d'Encapsulation

Un objet regroupe ses attributs (variables) et ses méthodes (code) capables d'interagir dessus. De plus, sa structure n'est pas visible de l'extérieur (**Abstraction de données**), ce qui permet de pouvoir modifier la structure interne de la classe sans que cela ne change quoi que ce soit pour l'utilisateur.

L'interaction avec les objets se fait via les méthodes propres à ce dernier, ainsi on n'a aucune information sur le déroulement interne, si le traitement demandé a fait appel a plusieurs méthodes ou encore s'il a demandé la création d'objet, c'est ce qu'on appelle **l'abstraction procédurale**.

Cependant les langages objets n'obligent pas à respecter le principe d'encapsulation, c'est au programmeur de faire attention.

2. Les Classes

2.1.Création d'une classe

Pour créer une application en Java, il faut créer une nouvelle classe. En créant une classe, on crée un nouveau type de variable. En Java, il existe deux types de variables :

Les types primitifs : int, float, boolean, …

Les types de références (des classes) : **Object**, **String** (chaîne de caractères) , **System**, …

La JRE fournie de nombreuses classes, donc de nombreux types de références.

Le code définissant une classe est contenu dans un fichier texte ayant l'extension **.java**. ; après compilation nous obtenons un fichier **.class** contenant le *byte-code* de cette classe.

Un fichier Java peut contenir plusieurs classes (déconseillé), cependant il ne peut y en avoir qu'une seule classe déclarée publique. Les autres classes seront sans modificateur de visibilité (default).

La classe déclarée publique doit avoir exactement le même nom que le fichier.

On utilise le mot clé **class** pour déclarer la classe que l'on veut créer.

Exemple :

```
//nom du fichier Exemple.java

public class Exemple {
	// code de la classe
}
```

2.2.Le constructeur

2.2.1. Définition

Le constructeur est une méthode d'une classe qui permet de créer un objet de cette classe. Le constructeur définit les tâches à effectuer lors de la création du nouvel objet. Un constructeur possède deux caractéristiques :

Il doit porter exactement le même nom que la classe à laquelle il appartient (faites attention à respecter la casse, majuscule/minuscule)

Il ne peut retourner aucune valeur (y compris **void**)

Un constructeur dépourvu d'argument (ne prenant pas de paramètre) est défini par Java comme un constructeur par défaut. Il n'est pas obligatoire de définir un constructeur de manière explicite dans sa classe. En effet, il existe toujours un constructeur par défaut (sans argument) défini de manière implicite. Vous n'êtes pas obligé de définir de constructeur, si vous estimez qu'aucune action spécifique n'est à effectuer lors de la création de l'objet.

2.2.2. Surdéfinition du constructeur

De manière générale, la surdéfinition consiste à redéfinir une méthode de manière explicite (utilisé pour l'héritage). Si vous déclarez un constructeur dans votre classe, vous remplacez alors celui par défaut (implicite) par ce nouveau (explicite).

Exemple :

```
// fichier Personne.java

public class Personne {

	Personne() { // constructeur par défaut de la classe personne
		     // déclaré de façon explicite
	}
}
```

2.2.3. Surcharge du constructeur

Un constructeur (comme tout autre méthode) peut être surchargé, en fonction des paramètres passés, l'interpréteur va appeler le constructeur qu'il faut.

Exemple :

```
// fichier Personne.java

public class Personne {

	Personne() {
		// constructeur de la classe personne (constructeur  par défaut)
	}

	Personne(String nom) {
		//surcharge
	}

	Personne(String nom, String adresse) {
		//surcharge
	}
}
```

2.3.Instancier une classe

2.3.1. Définition

On appelle instanciation le fait de créer un nouvel objet. L'instanciation consiste donc en une allocation mémoire. On dit qu'un objet est une instance d'une classe. Généralement, on entend par instanciation, la création de l'objet et aussi son association à un nom de variable (affectation de la variable).

2.3.2. L'opérateur new

L'opérateur **new** spécifie que l'on veut créer un nouvel objet. Celui-ci est suivi de l'appel du constructeur correspondant à la classe que l'on veut instancier.

Exemple :

```
class Exemple {

}
new Exemple();
```

L'objet est bien créé cependant il représente peu d'intérêt puisque, ne possédant pas de nom. Il n'est plus manipulable dès lors qu'il a été créé.

2.3.3. Affectation d'objets

On peut identifier les objets grâce une référence (ou identificateur) qu'on va lui donner au moment de son instanciation. On va faire pointer la référence vers l'objet nouvellement créé en utilisant l'opérateur égal "=". Cela s'appelle l'affectation.

Exemple :

```
class Exemple {

}

Exemple ex1 = new Exemple();

//ou

Exemple ex2;
ex2 = new Exemple();
```

Exemple ex1	=	*new*	Exemple();
Déclaration	affectation	*Instanciation*	Initialisation

On peut également affecter un objet à un autre objet.

Exemple :

```
Exemple ex1 = new Exemple() ;
Exemple ex2 = new Exemple() ;
ex1=ex2 ;
```

Dans la partie gauche de la ligne 1, on déclare une nouvelle variable ex1 de type Exemple (Exemple ex1). ex1 peut donc référencer tout objet de type Exemple. De l'autre côté, on instancie un objet Exemple qu'on appellera A. L'opérateur sert à spécifier que l'objet A nouvellement créé est référencé par la variable ex1.

A la ligne 2, le principe reste le même : On crée un objet Exemple qu'on appellera B qui est référencé par la variable ex2.

A la ligne 3, on décide que ex1 fait maintenant référence à l'objet B. ex1 et ex2 font référence au même objet. L'objet A qui était référencé par ex1 n'est alors plus référencé.

2.4.La méthode main

2.4.1. Définition

La méthode main permet l'exécution du programme. C'est la première méthode qui est appelée lors de l'exécution de la classe. On dit que c'est le point d'entrée du programme. Cette méthode est similaire à celle du C/C++.

2.4.2. Contraintes

La méthode main doit être dans une classe publique.

Elle doit elle même être déclarée en publique (accessible par l'interpréteur java).

Cette méthode doit être statique (pas besoin de créer d'objet pour l'appeler).

Elle prend obligatoirement en paramètre un tableau de chaîne de caractères (**String**).

Enfin, cette méthode doit renvoyer un type de données **void**.

Exemple :

```
//nom du fichier Exemple.java

public class Exemple {

	public static void main(String[] args) {

	}
}
```

2.4.3. Paramètres

Le paramètre **« String[] args »** de la méthode **main** va permettre de récupérer les paramètres de la ligne de commande en les stockant dans le tableau de chaîne de caractères (**String**) appelé args. Ce tableau peut avoir le nom que l'on veut. Par convention, on l'appel args.

Exemple :

```
//nom du fichier Exemple.java

public class Exemple {

	public static void main(String[] args) {
		System.out.println(args[0]);
		System.out.println(args[1]);
	}
}

//ligne de commande sous un prompt
java Exemple argument1 argument2

//le programme affiche
argument1
argument2
```

2.5.L'opérateur : "." et le pointeur "this"

2.5.1. L'opérateur "."

Il permet l'accès aux ressources et méthodes d'une instance, comme dans la majorité des langages objets. Cependant l'accès est soumis à plusieurs règles (on ne peut pas accéder à n'importe quelles ressources ou méthodes d'une instance à n'importe quel endroit du code, voir le chapitre sur la « Visibilité »).

Principe d'utilisation :

nomInstance . *nomVariale*

nomInstance. *nomMéthode()*

Exemple :

```
public class Exemple {

	int a;

	public String getText(){
		return "Hello World";
	}

	public void setText(String s){

	}
}

//création de l'instance
Exemple ex1 = new Exemple();

//acces
ex1.a; // accès à la variable int a
ex1.getText(); // retourne un string
ex1.setText("Hello"); // envoie un String par la méthode setText()
```

2.5.2. Le pointeur "this"

this est le seul pointeur en Java, il va permettre à une méthode d'accéder à la classe dans laquelle elle est définie.

Par défaut, dans une méthode, Java fait référence à la variable de la classe. Cependant si une variable dans cette même méthode porte le même nom, Java va faire référence à la variable de la méthode, grâce à **this** on va pouvoir préciser la variable que l'on souhaite.

Exemple :

```
public class ExempleThis {

	String s = "Variable de la classe";		//variable de la classe

public static void main(String [] args) {
	new ExempleThis().appel();
```

```
	}

	public void appel() {
		System.out.println(s);		//Affichage de la valeur de s
		String s = "Variable de la méthode";
		System.out.println(s);
		System.out.println(this.s);
	}
}
```

L'affichage résultant donne :

```
Variable de la classe
Variable de la méthode
Variable de la classe
```

On peut aussi utiliser la méthode **this()** pour appeler un constructeur d'une classe, cette méthode peut être seulement appelé au début d'un constructeur. On l'utilise dans un constructeur pour appeler un autre constructeur (surchargé).

Exemple :

```
public class ExempleThis {

	int a;

	public ExempleThis() {
		this(217);
	}

	public ExempleThis(int b) {
		a = b;
	}
}
```

Dans cet exemple **this(217)** va appeler le constructeur **public ExempleThis(int b)**.

3. Utilisation des packages

3.1.Package

3.1.1. Définition

Les packages permettent de regrouper des classes ayant des caractéristiques communes. On peut considérer un package comme une librairie ou une partie de librairie. Par exemple, les classes permettant de travailler avec des flux d'entrée/sortie sont dans le package **java.io**. En réalité, un package est un répertoire physique sur votre système. Un nom de package est généralement un nom composé. Chaque mot formant le nom du package sont séparés par des points.

Prenons l'exemple de la classe "Tools" appartenant au package "com.sun.java". Au niveau du système, on aura un répertoire com contenant un répertoire sun, contenant lui-même un répertoire java. Le fichier Tools.class sera placé dans le répertoire java.

3.1.2. Utilisation

Au niveau du fichier, la déclaration du package se place en premier. Afin d'éviter que plusieurs personnes utilisent le même nom de package, Sun recommande d'utiliser l'inverse de son nom de domaine pour nommer ses packages (les packages venant de Sun s'appelleront com.sun.java).

Exemple :

```
//fichier Tools.java
package com.sun.java;

// La classe Tools va être situé dans le package com.sun.java
public class Tools {
	//votre code
}
```

On crée un répertoire "src" qui contiendra nos fichiers sources (.java). Dans notre exemple, il s'agirat simplement du fichier Tools.java. On crée également un répertoire "bin" qui contiendra les bytes codes (.class). Lors de la compilation, on spécifiera ces deux répertoires à javac.

```
javac –d c:\java\bin –sourcepath c:\java\src com.sun.java.Tools.java
```

On peut remarquer que javac a crée le fichier c:\java\bin\com\sun\java\Tools.class ainsi que les répertoires parents.

Pour l'exécution, on rajoute à la variable d'environnement CLASSPATH *(chemin des classes)* le chemin du répertoire bin. Le classpath permet à l'interpréteur de savoir où sont situés physiquement les fichiers contenant le byte code.

Exemple sous Unix :

```
set CLASSPATH=/home/myaccount/lib
export CLASSPATH
```

Exemple sous Windows :

```
set CLASSPATH=C:\Java\lib
```

On exécute en utilisant java.exe :

```
java –classpath c:\java\lib com.sun.java.Tools
```

3.2.import

Le mot clé **import** permet d'importer (d'accéder et d'utiliser) des classes d'un package à partir d'une classe d'un autre package. Il est possible d'importer toutes les classes d'un package grâce au caractère étoile "*". L'importation se fait au début du code entre la déclaration du package et de la classe.

```
//fichier Ex.java
package com.sun.exemple;

import com.sun.java.Tools;
//importe la classe Tools du package com.sun.java

import java.utils.Vector;
//importe la classe Vector du package java.utils

import com.sun.*;
//importe toutes les classes du package com.sun

public class Ex {
      Tools tools = new Tools();
      // ayant importé Tools, on peut l'utiliser
}
```

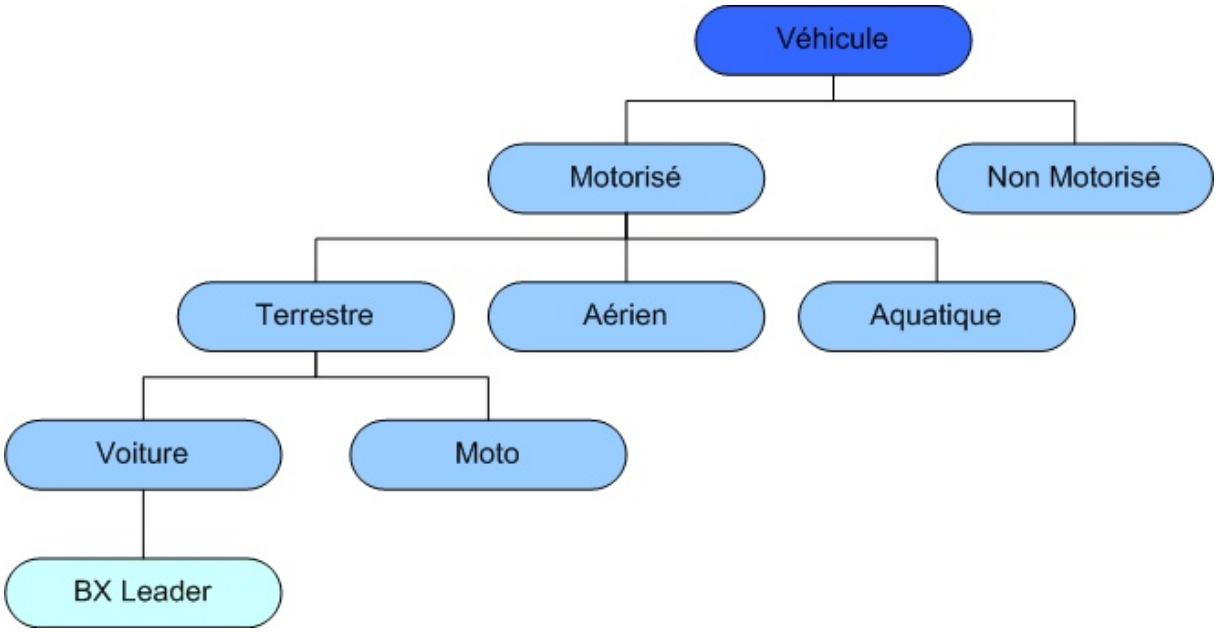
4. Héritage

4.1.Définition

L'héritage est la capacité de construire une classe à partir d'une autre, cette dernière héritera des variables et méthodes de la classe de base, et pourra y accéder suivant l'accessibilité définie.

Java se démarque du C++ ou d'autres langages objet puisque il ne propose pas d'héritage multiple. Cependant il existe un moyen de contourner le problème grâce aux interfaces.

Prenons l'exemple d'une voiture « BX Leader » (ici BX Leader est un objet de la classe voiture), voici ce que pourrait donner la hiérarchie des classes. Ceci n'est qu'un exemple, il n'a y a pas de solution préétablie, il faut juste que l'ensemble soit cohérant et utile. Il n'y pas d'intérêt à faire une telle hiérarchie si on utilise un seul type de véhicule (par exemple seulement les voitures).



Les classes parentes sont appelées super-classe, les classes dérivées sont appelées sous-classes ou classes filles. Dans l'exemple précèdent, "Véhicule" est la super-classe, "Motorisé" la sous-classe, etc.

En Java tout est objet. En effet, tout classe hérite de la classe **Object**. La classe **Object** est donc à la racine de toute hiérarchie. On dit parfois que c'est la classe **racine**.

4.2.extends

L'héritage se fait via le mot clé **extends,** il se place après le nom de la classe, au moment de la définition de cette dernière, et est suivit du nom de la classe à hériter.

Exemple :

```
class Personne {    //super-classe

}
class Etudiant extends Personne {      //sous-classe

}
```

4.3.super

Le mot clé **super** est similaire à **this**. Contrairement au mot clé this qui permet de faire référence à la classe dans laquelle il est appelé, le mot clé super fait référence à la super-classe de la classe dans laquelle il est appelé.

Il permet d'invoquer le constructeur de la super-classe. **super** doit être déclaré au tout début d'un constructeur d'une sous-classe. En effet, cela permet au constructeur d'effectuer au moins les mêmes tâches que le constructeur de la classe mère.

Exemple :

```
class Personne {

	Personne(String nom) {
		//constructeur
	}
}

class Etudiant extends Personne {

	int note;

	Etudiant(String nom, int note) {
		super(nom); // constructeur de la classe mère
		this.note = note;
	}
}
```

Dans cet exemple ***super()*** fait appel au constructeur de la super-classe **Personne**.

5. Les interfaces (héritage multiple)

5.1. Définition

Une interface est très similaire d'une classe. Voici les différences par rapport à une classe :

- elle ne possède que des méthodes sans corps
- elle ne possède pas de constructeur
- elle n'est pas instanciable
- ses variables doivent être initialisées et sont implicitement publiques (**public**), statiques (**static**) et finales (**final**)

En fait, une interface est un peu le squelette de la classe, par conséquent toutes les méthodes abstraites qu'elle possède devront être redéfinies dans la classe. Une interface peut hériter (utilisation de **extends**) d'une ou plusieurs autres interfaces.

Exemple :

```
interface Math extends Serializable {

	float pi = 3.14;	// pi doit obligatoirement être initialisé

	float perimetre(float a);
}
```

5.2. Implémentation

L'implémentation d'interface permet de palier à l'absence d'héritage multiple. Une classe peut implémenter plusieurs interfaces. En implémentant une interface, une classe hérite de ses variables ainsi que de ses méthodes qu'elle devra définir (sauf en cas d'utilisation du mot clé **abstract**).

Elle se fait grâce au mot clé **implements**.

Exemple :

```
class Cercle implements Math, Interface1 {

	float perimetre(float rayon) {	// On définit le corps de la méthode
		return pi * rayon * 2;	// pi est hérité de Math
	}
}

class Exemple implements Interface1, Interface2, Interface3 {

}
```

5.3. Comparaison avec les classes abstraites

Il est parfois difficile de faire la différence entre une classe abstraite et une interface. L'avantage d'une classe abstraite est qu'elle peut contenir des méthodes classiques (non abstraites). Une interface ne

contient que des méthodes sans corps qui devront être défini par les classes qui l'implémente. L'avantage des interfaces est qu'il est possible d'implémenter plusieurs interfaces, or on ne peut hériter de plusieurs classes.

6. Modificateurs de visibilité

6.1.public

Accessible par toutes les classes de tous les packages. Ce modificateur peut s'appliquer aux classes, méthodes et variables membres de classe. Certaines classes doivent être **public** pour permettre l'exécution de l'application (celles contenant une méthode ***main()***).

6.2.private

Ce modificateur peut s'appliquer aux méthodes et variables membres de classe ainsi qu'aux *inner-classes* (classes contenues dans d'autres classes). Les classes de premier niveau (*top-level/outer classes*, qui ne sont pas contenues dans d'autres classes) ne peuvent bénéficier de cet attribut. Ces éléments, lorsqu'ils sont définis avec l'attribut **private**, ne sont pas accessibles à partir d'autres classes ni par les sous-classes.

6.3.protected

Comme **private**, ce modificateur peut s'appliquer aux méthodes, variables membres et *inner-classes*. Les éléments définis avec ce modificateur sont accessibles par toutes les classes dérivées et les classes du même package.

L'attribut **protected** permet l'accès à la ressource par les classes dérivés (ces dernières peuvent être situées dans un autre package) et toutes les classes du même package, comme le montre l'exemple suivant :

```
//Fichier (..)\exa\Mere.java
package exa;

public class Mere {

	protected int tdb;

}
```

```
//Fichier (..)\exb\Fille.java
package exb;

import exa;

class Fille extends Mere {

	public static void main(String[] args) {
		tdb++; // accès autorisé

		Fille objf ;
		objf.tdb++;//accès autorisé

		Mere objm;
		objm.tdb++; //accès interdit, erreur de compilation
	}
}
```

6.4.Aucun modificateur

Lorsque vous ne spécifiez aucun modificateur particulier, Java utilise le modificateur de visibilité par défaut. Les variables, méthodes et classes déclarées ainsi sont accessibles par les classes qui font partie du même package et inaccessible par les autres (équivalent du « friend » en C++).

7. Autres modificateurs

7.1. abstract

7.1.1. Définition

Une classe abstraite ne peut être instanciée. Les classes abstraites servent pour l'héritage comme super-classe. Contrairement aux classes classiques, elles peuvent contenir des méthodes abstraites.

Les méthodes abstraites sont des squelettes de méthodes. Elles n'ont pas de corps (aucune ligne de code). Une méthode abstraite étant vide, elle devra donc être surdéfinie (polymorphisme) par les classes filles.

7.1.2. Utilisation

Les classes ou méthodes abstraites sont déclarées avec le mot clé **abstract**.

Exemple :

```
public abstract class Test {

	public abstract void methodeAbstraite();

	public void methodeClassique() {

	}

}
```

On ne peut pas avoir une méthode **abstract** dans une classe qui n'est pas elle-même déclarée en abstract.

Exemple :

```
public class IllegalClass { // abstract n'est pas spécifié

	public abstract void illegalMethod(); // erreur de compilation

}
```

Si vous essayez de compiler ce code vous aurez une erreur de type : "class IllegalClass must be declared abstract".

7.1.3. Intérêt

Une sous classe qui hérite d'une classe abstraite devra obligatoirement surdéfinir les méthodes abstraites de sa classe mère (sauf si elle est elle-même déclarée abstraite). L'intérêt étant que chaque sous classes devront définir ces méthodes.

Exemple :

```
// Classe mère des animaux
public abstract class Animal {
```

```
	public abstract void seDeplacer();
// tous les animaux peuvent se déplacer mais pas forcément de la même manière

}

// Classe Tigre héritant d'Animal
public class Tigre extends Animal {

	public void seDeplacer() {
		courrir();
	}

	public void courrir() {
		// code
	}

}
```

```
// Classe Aigle héritant d'Animal
public class Aigle extends Animal {

	public void seDeplacer() {
		voler();
	}

	public void voler() {
		// code
	}

}
```

```
// Classe Application
public class Application {

	public static void main(String[] args) {
		Animal animal1 = new Tigre();
		Animal animal2 = new Aigle();
		animal1.seDeplacer();
		animal2.seDeplacer();
	}
}
```

Dans cette exemple, la classe **Animal** déclare une méthode ***seDeplacer()*** qui devra être définie par toutes les sous classes. Cette méthode est abstraite car tous les animaux se déplace mais pas de la même manière. C'est donc aux sous classes de décrire la manière de se déplacer. Cette exemple, montre l'utilisation du polymorphisme : on manipule animal1 et animal2 comme de simple animaux, sans forcément connaître à quel type exact d'animaux ils appartiennent. On sait simplement que ce sont des animaux donc ils possèdent une méthode ***seDeplacer()***.

7.2.final

Le mot clé **final** sert à spécifier ce qui ne peut être modifié. Ils peut être associé à une variable, une méthode ou une classe.

7.2.1. Variable finale

Une variable finale ne peut être modifiée. On lui affecte une valeur lors de sa déclaration. Une fois que l'objet est crée (appel du constructeur), on ne plus changer cette valeur. Une variable est en quelque sorte une constance.

Exemple :

```
public class Math {

	public final float pi = 3.14;

}
```

7.2.2. Méthode finale

Une méthode finale ne peut être surdéfinie dans une classe fille.

Exemple :

```
public class Exemple {

	public final void uneMethode(){
	}
}
```

7.2.3. Classe finale

Une classe finale ne peut pas être dérivé et ne peut pas être abstraite. L'intérêt des classes finales résident dans un soucis d'optimisation et de sécurité. Toutes classes en bout de hiérarchie devraient être déclarées finales.

Exemple :

```
final class Exemple {

}
```

7.3.static

Le mot clé **static (**vu avec la méthode main) peut être associé à une variable ou une méthode. Il permet d'accéder à cette ressource sans avoir à créer d'objet.

Exemple :

```
public class Math {
	static float pi = 3.14;

	static float perimetre(float largeur, float longueur) {
		return ((largeur+longueur)*2);
	}
}
```

```
public class Exemple {
	public static void main(String[] args) {
		float p = Math.pi;
		p = Math.perimetre(3, 4);
	}
}
```

Dans cet exemple, on accède à la variable pi et à la méthode perimetre() sans avoir crée d'objet de type Math.

Une autre particularité est qu'une variable statique aura la même valeur pour toutes les instances (objets) de la classe. Il faut donc faire très attention lorsqu'on modifie la valeur d'une variable statique. C'est pour cette raison que les constantes sont déclarés static et final.

Exemple :

```
public class Exemple2 {

	public static void main(String[] args) {
		Math m1 = new Math();
		m1.pi = 4;
		Math m2 = new Math();
		float p = m2.pi; // p vaut 4
	}
}
```

8. Garbage collector

8.1. Rôle

Contrairement au C++, Java ne possède pas de destructeur, le garbage gollector (ou ramasse miette) s'occupe de détruire les instances qui ne sont plus référencées. Le garbage collector surveille toutes les instances, détecte celles qui sont inutiles et les retire de la mémoire. Cependant, juste avant de détruire les instances il va appeler la méthode **finalize()**, si elle existe.

8.2. Méthode finalize()

La méthode **finalize** est donc appelée par le garbage collector juste avant la destruction d'une instance, cela permet par exemple de libérer les ressources que l'instance utilise ou encore d'exécuter certaines tâches à la destruction de l'instance. La méthode doit être écrite par le développeur, il faut donc la définir dans la classe.

Exemple :

```
public class Exemple {

	public void finalize() throws Exception {
		// faire attention aux exceptions (voir
		// cours sur les exceptions)
	}
}
```

8.3. Contrôle du garbage collector

Le garbage collector est entièrement intégrée à la JVM (Java Virtual Machine). Il intervient lorsqu'il y a un manque de la mémoire. Il est impossible de contrôler son passage. Cependant, on peut forcer son intervention grâce à la méthode statique ***gc()*** de la classe **System**.