La programmation orientée objet

L'idée de base de la programmation orientée objet est de rassembler dans une même entité appelée objet les données et les traitements qui s'y appliquent.

Ce cours contient plusieurs sections :

- Le concept de classe : présente le concept et la syntaxe de la déclaration d'une classe
- Les objets : présente la création d'un objet, sa durée de vie, le clonage d'objets, les références et la comparaison d'objets, l'objet null, les variables de classes, la variable this et l'opérateur instanceof.
- Les modificateurs d'accès : présente les modificateurs d'accès des entités classes, méthodes et attributs ainsi que les mots clés qui permettent de qualifier ces entités
- Les propriétés ou attributs : présente les données d'une classe : les propriétés ou attributs
- Les méthodes : présente la déclaration d'une méthode, la transmission de paramètres, l'émission de messages, la surcharge, la signature d'une méthode et le polymorphisme et des méthodes particulières : les constructeurs, le destructeur et les accesseurs
- L'héritage : présente l'héritage : son principe, sa mise en oeuvre, ses conséquences. Il présente aussi la redéfinition d'une méthode héritée et les interfaces
- Les packages : présente la définition et l'utilisation des packages
- Les classes internes : présente une extension du langage Java qui permet de définir une classe dans une autre.
- La gestion dynamique des objets : présente rapidement la gestion dynamique des objets grâce à l'introspection

I. Le concept de classe

Une classe est le support de l'encapsulation : c'est un ensemble de données et de fonctions regroupées dans une même entité. Une classe est une description abstraite d'un objet. Les fonctions qui opèrent sur les données sont appelées des méthodes. Instancier une classe consiste à créer un objet sur son modèle. Entre classe et objet il y a, en quelque sorte, le même rapport qu'entre type et variable.

Java est un langage orienté objet : tout appartient à une classe sauf les variables de types primitives.

Pour accéder à une classe il faut en déclarer une instance de classe ou objet.

Une classe comporte sa déclaration, des variables et les définitions de ses méthodes.

Une classe se compose de deux parties : un en−tête et un corps. Le corps peut être divisé en 2 sections : la déclaration des données et des constantes et la définition des méthodes. Les méthodes et les données sont pourvues d'attributs de visibilité qui gèrent leur accessibilité par les composants hors de la classe.

1. La syntaxe de déclaration d'une classe

La syntaxe de déclaration d'une classe est la suivante :

modificateurs class nom_de_classe [extends classe_mere] [implements interfaces] { ... }

Les modificateurs de classe sont :

Modificateur	Rôle
abstract	la classe contient une ou des méthodes abstraites, qui n'ont pas de définition explicite. Une classe déclarée abstract ne peut pas être instanciée : il faut définir une classe qui hérite de cette classe qui implémente les méthodes nécessaires pour ne plus être abstraite.
final	la classe ne peut pas être modifiée, sa redéfinition grâce à l'héritage est interdite. Les classes déclarées final ne peuvent donc pas avoir de classes filles.
private	la classe n'est accessible qu'à partir du fichier où elle est définie
public	La classe est accessible partout

Les modificateurs abstract et final ainsi que public et private sont mutuellement exclusifs.

Marquer une classe comme final peut permettre au compilateur et à la JVM de réaliser quelques petites optimisations.

Le mot clé extends permet de spécifier une superclasse éventuelle : ce mot clé permet de préciser la classe mère dans une relation d'héritage.

Le mot clé implements permet de spécifier une ou des interfaces que la classe implémente. Cela permet de récupérer quelques avantages de l'héritage multiple.

L'ordre des méthodes dans une classe n'a pas d'importance. Si dans une classe, on rencontre d'abord la méthode A puis la méthode B, B peut être appelée sans problème dans A.

II. Les objets

Les objets contiennent des attributs et des méthodes. Les attributs sont des variables ou des objets nécessaires au fonctionnement de l'objet. En Java, une application est un objet. La

classe est la description d'un objet. Un objet est une instance d'une classe. Pour chaque instance d'une classe, le code est le même, seules les données sont différentes à chaque objet.

1. La création d'un objet : instancier une classe

Il est nécessaire de définir la déclaration d'une variable ayant le type de l'objet désiré. La déclaration est de la forme nom_de_classe nom_de_variable

Exemple :

MaClasse m;

String chaine;

L'opérateur new se charge de créer une instance de la classe et de l'associer à la variable

Exemple :

m = new MaClasse();

Il est possible de tout réunir en une seule déclaration

Exemple : *MaClasse m = new MaClasse();*

Chaque instance d'une classe nécessite sa propre variable. Plusieurs variables peuvent désigner un même objet.

En Java, tous les objets sont instanciés par allocation dynamique. Dans l'exemple, la variable m contient une référence sur l'objet instancié (contient l'adresse de l'objet qu'elle désigne : attention toutefois, il n'est pas possible de manipuler ou d'effectuer des opérations directement sur cette adresse comme en C).

Si m2 désigne un objet de type MaClasse, l'instruction m2 = m ne définit pas un nouvel objet mais m et m2 désignent tous les deux le même objet.

L'opérateur new est un opérateur de haute priorité qui permet d'instancier des objets et d'appeler une méthode particulière de cet objet : le constructeur. Il fait appel à la machine virtuelle pour obtenir l'espace mémoire nécessaire à la représentation de l'objet puis appelle le constructeur pour initialiser l'objet dans l'emplacement obtenu. Il renvoie une valeur qui référence l'objet instancié.

Si l'opérateur new n'obtient pas l'allocation mémoire nécessaire, il lève l'exception OutOfMemoryError.

2. La durée de vie d'un objet

Les objets ne sont pas des éléments statiques et leur durée de vie ne correspond pas forcément à la durée d'exécution du programme.

La durée de vie d'un objet passe par trois étapes :

la déclaration de l'objet et l'instanciation grâce à l'opérateur new

Exemple : nom_de_classe nom_d_objet = new nom_de_classe(...);

l'utilisation de l'objet en appelant ses méthodes

la suppression de l'objet : elle est automatique en Java grâce à la machine virtuelle. La restitution de la mémoire inutilisée est prise en charge par le récupérateur de mémoire (garbage collector). Il n'existe pas d'instruction delete comme en C++.

3. Les références et la comparaison d'objets

Les variables de type objet que l'on déclare ne contiennent pas un objet mais une référence vers cet objet. Lorsque l'on écrit c1 = c2 (c1 et c2 sont des objets), on copie la référence de l'objet c2 dans c1. c1 et c2 font référence au même objet : ils pointent sur le même objet. L'opérateur == compare ces références. Deux objets avec des propriétés identiques sont deux objets distincts :

Exemple :

```
Rectangle r1 = new Rectangle(100,50);
Rectangle r2 = new Rectangle(100,50);
if (r1 == r1) { ... } // vrai
if (r1 == r2) { ... } // faux
```

Pour comparer l'égalité des variables de deux instances, il faut munir la classe d'une méthode à cet effet : la méthode equals() héritée de Object.

Pour s'assurer que deux objets sont de la même classe, il faut utiliser la méthode getClass() de la classe Object dont toutes les classes héritent.

Exemple : *(obj1.getClass().equals(obj2.getClass())*

4. Le littéral null

Le littéral null est utilisable partout où il est possible d'utiliser une référenc à un objet. Il n'appartient pas à une classe mais il peut être utilisé à la place d'un objet de n'importe quelle type ou comme paramètre. null ne peut pas être utilisé comme un objet normal : il n'y a pas d'appel de méthodes et aucune classe ne peut en hériter.

Le fait d'affecter null une variable référençant un objet pourra permettre au ramasse–miettes de libérer la mémoire allouée à l'objet si aucune autre référence n'existe encore sur lui.

5. Les variables de classes

Elles ne sont définies qu'une seule fois quel que soit le nombre d'objets instanciés de la classe. Leur déclaration est accompagnée du mot clé static

Exemple :

```
public class MaClasse() {
static int compteur = 0;
}
```

L'appartenance des variables de classe à une classe entière et non à un objet spécifique permet de remplacer le nom de la variable par le nom de la classe.

Exemple :

```
MaClasse m = new MaClasse();
int c1 = m.compteur;
int c2 = MaClasse.compteur;
// c1 et c2 possèdent la même valeur.
```

Ce type de variable est utile pour, par exemple, compter le nombre d'instanciations de la classe.

6. La variable this

Cette variable sert à référencer dans une méthode l'instance de l'objet en cours d'utilisation. this est un objet qui est égal à l'instance de l'objet dans lequel il est utilisé.

Exemple :

private int nombre;

public maclasse(int nombre) {

nombre = nombre; // variable de classe = variable en paramètre du constructeur

}

Il est préférable de préfixer la variable d'instance par le mot clé this.

Exemple : *this.nombre = nombre;*

Cette référence est habituellement implicite :

Exemple :

```
class MaClasse() {
String chaine = " test " ;
public String getChaine() { return chaine; }
// est équivalent à public String getChaine() { return this.chaine; }
}
```

This est aussi utilisé quand l'objet doit appeler une méthode en se passant lui−même en paramètre de l'appel.

7. L'opérateur instanceof

L'opérateur instanceof permet de déterminer la classe de l'objet qui lui est passé en paramètre. La syntaxe est objet instanceof classe

Exemple :

```
void testClasse(Object o) {
if (o instanceof MaClasse )
System.out.println(" o est une instance de la classe MaClasse "); else
System.out.println(" o n'est pas un objet de la classe MaClasse ");
}
```

Dans le cas ci−dessus, même si o est une instance de MaClasse, il n'est pas permis d'appeler une méthode de MaClasse car o est de type Objet.

Exemple :

```
void afficheChaine(Object o) {
if (o instanceof MaClasse)
System.out.println(o.getChaine());
            erreur à la compil car la méthode getChaine()
        //n'est pas définie dans la classe Object
}
```

Pour résoudre le problème, il faut utiliser la technique du casting (conversion).

Exemple :

```
void afficheChaine(Object o) {
```

```
if (o instanceof MaClasse) {
MaClasse m = (MaClasse) o;
System.out.println(m.getChaine());
// OU System.out.println( ((MaClasse) o).getChaine() );
}
}
```

III. Les modificateurs d'accès

Ils s'appliquent aux classes, aux méthodes et aux attributs.

Ils ne peuvent pas être utilisés pour qualifier des variables locales : seules les variables d'instances et de classes peuvent en profiter.

Ils assurent le contrôle des conditions d'héritage, d'accès aux éléments et de modification de données par les autres objets.Les mots clés qui gèrent la visibilité des entités

De nombreux langages orientés objet introduisent des attributs de visibilité pour réglémenter l'accès aux classes et aux objets, aux méthodes et aux données.

En plus de la valeur par défaut, il existe 3 modificateurs explicites qui peuvent être utilisés pour définir les attributs de visibilité des entités (classes, méthodes ou attributs) : public, private et protected. Leur utilisation permet de définir des niveaux de protection différents (présentés dans un ordre croissant de niveau de protection offert) :

Modificateur	Rôle
public	Une variable, méthode ou classe déclarée public est visible par tous les autres objets. Depuis la version 1.0, une seule classe publique est permise par fichier et son nom doit correspondre à celui du fichier. Dans la philosophie orientée objet aucune donnée d'une classe ne devrait être déclarée publique : il est préférable d'écrire des méthodes pour la consulter et la modifier
par défaut :	

package−private	Il n'existe pas de mot clé pour définir ce niveau, qui est le niveau par défaut lorsqu'aucun modificateur n'est précisé. Cette déclaration permet à une entité (classe, méthode ou variable) d'être visible par toutes les classes se trouvant dans le même package.
protected	Si une méthode ou une variable est déclarée protected, seules les méthodes présentes dans le même package que cette classe ou ses sous−classes pourront y accéder. On ne peut pas qualifier une classe avec protected.
private	C'est le niveau de protection le plus fort. Les composants ne sont visibles qu'à l'intérieur de la classe : ils ne peuvent être modifiés que par des méthodes définies dans la classe et prévues à cet effet. Les méthodes déclarées private ne peuvent pas être en même temps déclarées abstract car elles ne peuvent pas être redéfinies dans les classes filles.

Ces modificateurs d'accès sont mutuellement exclusifs.

1. Le mot clé static

Le mot clé static s'applique aux variables et aux méthodes.

Les variables d'instance sont des variables propres à un objet. Il est possible de définir une variable de classe qui est partagée entre toutes les instances d'une même classe : elle n'existe donc qu'une seule fois en mémoire. Une telle variable permet de stocker une constante ou une valeur modifiée tour à tour par les instances de la classe. Elle se définit avec le mot clé static.

Exemple :

```
public class Cercle {
static float pi = 3.1416f;
float rayon;
public Cercle(float rayon) { this.rayon = rayon; } public
float surface() { return rayon * rayon * pi;}
}
```

Il est aussi possible par exemple de mémoriser les valeurs min et max d'un ensemble d'objets de même classe.

Une méthode static est une méthode qui n'agit pas sur des variables d'instance mais uniquement sur des variables de classe. Ces méthodes peuvent être utilisées sans instancier un objet de la classe. Les méthodes ainsi définies peuvent être appelées avec la notation classe.methode() au lieu de objet.methode() : la première forme est fortement recommandée pour éviter toute confusion.

Il n'est pas possible d'appeler une méthode d'instance ou d'accéder à une variable d'instance à partir d'une méthode de classe statique.Le mot clé final

Le mot clé final s'applique aux variables de classe ou d'instance ou locales, aux méthodes, aux paramètres d'une méthode et aux classes. Il permet de rendre l'entité sur laquelle il s'applique non modifiable une fois qu'elle est déclarée pour une méthode ou une classe et initialisée pour une variable.

Une variable qualifiée de final signifie que la valeur de la variable ne peut plus être modifiée une fois que celle–ci est initialisée.

Exemple :

```
package tg.epl.cours;
public class Constante2 {
public final int constante;
public Constante2() {
this.constante = 10;
}
}
```

Une fois la variable déclarée final initialisée, il n'est plus possible de modifier sa valeur. Une vérification est opérée par le compilateur.

Exemple :

```
package tg.epl.cours;
public class Constante1 {
public static final int constante = 0;
public Constante1() {
this.constante = 10;
}
}
```

Résultat :

C:\>javac Constante1.java

Constante1.java:6: cannot assign a value to final variable constante
this.constante = 10;
^
1 error

Les constantes sont qualifiées avec les modificateurs final et static.

Exemple : *public static final float PI = 3.141f;*

Une méthode déclarée final ne peut pas être redéfinie dans une sous−classe. Une méthode possédant le modificateur final pourra être optimisée par le compilateur car il est garanti qu'elle ne sera pas sous−classée.
Lorsque le modificateur final est ajouté à une classe, il est interdit de créer une classe qui en hérite.
Pour une méthode ou une classe, on renonce à l'héritage mais ceci peut s'avérer nécessaire pour des questions de sécurité ou de performance. Le test de validité de l'appel d'une méthode est bien souvent repoussé à l'exécution, en fonction du type de l'objet appelé (c'est la notion de polymorphisme qui sera détaillée ultérieurement). Ces tests ont un coût en termes de performance.Quatre types de variables sont implicitement déclarés final :

- un champ d'une interface
- une variable locale déclarée comme ressource d'une instruction try−with−resources
- un paramètre d'exception d'une clause multi−catch
- un champ correspondant à un composant d'un record

Remarque : un unique paramètre d'exception d'une clause catch n'est jamais déclaré final implicitement, mais peut être effectivement final.

2. Le mot clé abstract

Le mot clé abstract s'applique aux méthodes et aux classes.
Abstract indique que la classe ne pourra être instanciée telle quelle. De plus, toutes les méthodes de cette classe abstract ne sont pas implémentées et devront être redéfinies par des méthodes complètes dans ses sous−classes.
Abstract permet de créer une classe qui sera une sorte de moule. Toutes les classes dérivées pourront profiter des méthodes héritées et n'auront à implémenter que les méthodes déclarées abstract.

Exemple :
abstract class ClasseAbstraite {
ClasseAbstraite() { ... //code du constructeur }

```
void methode() { ... // code partagé par tous les descendants }
abstract void methodeAbstraite();
}
class ClasseComplete extends ClasseAbstraite {
ClasseComplete() { super(); ... }
void methodeAbstraite() { ... // code de la méthode } // void
methode est héritée
}
```

Une méthode abstraite est une méthode déclarée avec le modificateur abstract et sans corps. Elle correspond à une méthode dont on veut forcer l'implémentation dans une sous−classe. L'abstraction permet une validation du codage : une sous−classe sans le modificateur abstract et sans définition explicite d'une ou des méthodes abstraites génère une erreur de compilation.

Une classe est automatiquement abstraite dès qu'une de ses méthodes est déclarée abstraite. Il est possible de définir une classe abstraite sans méthodes abstraites.

3. Le mot clé synchronized

Il permet de gérer l'accès concurrent aux variables et méthodes lors de traitements de threads (exécution « simultanée » de plusieurs petites parties de code du programme)

4. Le mot clé volatile

Le mot clé volatile s'applique aux variables.
Il précise que la variable peut être changée par un périphérique ou de manière asynchrone. Cela indique au compilateur de ne pas stocker cette variable dans des registres. A chaque utilisation, sa valeur est lue et réécrite immédiatement si elle a changé.Le mot clé native
Une méthode native est une méthode qui est implémentée dans un autre langage. L'utilisation de ce type de méthode limite la portabilité du code mais permet une vitesse d'exécution plus rapide.

IV. Les propriétés ou attributs

Les données d'une classe sont contenues dans des variables nommées propriétés ou attributs. Ce sont des variables qui peuvent être des variables d'instances, des variables de classes ou des constantes.

1. Les variables d'instances

Une variable d'instance nécessite simplement une déclaration de la variable dans le corps de la classe.

Exemple :

```
public class MaClasse {
public int valeur1 ;
int valeur2 ;
```

protected int valeur3 ;
private int valeur4 ;
}

Chaque instance de la classe a accès à sa propre occurrence de la variable.

2. Les variables de classes

Les variables de classes sont définies avec le mot clé static

Exemple (code Java 1.1) :
public class MaClasse {
static int compteur ;
}

Chaque instance de la classe partage la même variable.

3. Les constantes

Les constantes sont définies avec le mot clé final : leur valeur ne peut pas être modifiée une fois qu'elles sont initialisées.

Exemple (code Java 1.1) :
public class MaClasse {
final double pi=3.14 ;
}

V. Les méthodes

Les méthodes sont des fonctions qui implémentent les traitements de la classe.

1. La syntaxe de la déclaration

La syntaxe de la déclaration d'une méthode est :
modificateurs type_retourné nom_méthode (arg1, ...) {... } // définition des variables locales et du bloc d'instructions
Le type retourné peut être élémentaire ou correspondre à un objet. Si la méthode ne retourne rien, alors on utilise void.
Le type et le nombre d'arguments déclarés doivent correspondre au type et au nombre d'arguments transmis. Il n'est pas possible d'indiquer des valeurs par défaut dans les paramètres. Les arguments sont passés par valeur : la méthode fait une copie de la variable qui lui est locale. Lorsqu'un objet est transmis comme argument à une méthode, cette dernière reçoit une référence qui désigne son emplacement mémoire d'origine et qui est une copie de la variable. Il est possible de modifier l'objet grâce à ses méthodes

mais il n'est pas possible de remplacer la référence contenue dans la variable passée en paramètre : ce changement n'aura lieu que localement à la méthode.
Les modificateurs de méthodes sont :

Modificateur	Rôle
public	la méthode est accessible aux méthodes des autres classes
private	l'usage de la méthode est réservé aux autres méthodes de la même classe
protected	la méthode ne peut être invoquée que par des méthodes de la classe ou de ses sous−classes
final	la méthode ne peut être modifiée (redéfinition lors de l'héritage interdite)
static	la méthode appartient simultanément à tous les objets de la classe (comme une constante déclarée à l'intérieur de la classe). Il est inutile d'instancier la classe pour appeler la méthode mais la méthode ne peut pas manipuler de variable d'instance. Elle ne peut utiliser que des variables de classes.
synchronized	la méthode fait partie d'un thread. Lorsqu'elle est appelée, elle barre l'accès à son instance. L'instance est à nouveau libérée à la fin de son exécution.
native	le code source de la méthode est écrit dans un autre langage

Sans modificateur, la méthode peut être appelée par toutes autres méthodes des classes du package auquel appartient la classe.
La valeur de retour de la méthode doit être transmise par l'instruction return. Elle indique la valeur que prend la méthode et termine celle−ci : toutes les instructions qui suivent return sont donc ignorées.

Exemple :

```
int add(int a, int b) {
return a + b;
}
```

Il est possible d'inclure une instruction return dans une méthode de type void : cela permet de quitter la méthode.
La méthode main() de la classe principale d'une application doit être déclarée de la façon suivante : public static void main (String args[]) { ... }

Exemple :

```
public class MonApp1 {
public static void main(String[] args) {
System.out.println("Bonjour");
}
}
```

Cette déclaration de la méthode main() est imposée par la machine virtuelle pour reconnaitre le point d'entrée d'une application. Si la déclaration de la méthode main() diffère, une exception sera levée lors de la tentative d'exécution par la machine virtuelle.

Exemple :

```
public class MonApp2 {
public static int main(String[] args) {
System.out.println("Bonjour");
return 0;
}
}
```

Résultat :

```
C:\>javac MonApp2.java
C:\>java MonApp2
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Si la méthode retourne un tableau alors les caractères [] peuvent être précisés après le type de retour ou après la liste des paramètres :

Exemple :

```
int[] getValeurs() { ... }
int getValeurs()[] { ... }
```

2. La transmission de paramètres

Lorsqu'un objet est passé en paramètre, ce n'est pas l'objet lui−même qui est passé mais une référence sur l'objet. La référence est bien transmise par valeur et ne peut pas être modifiée mais l'objet peut être modifié par un message (appel d'une méthode).

Pour transmettre des arguments par référence à une méthode, il faut les encapsuler dans un objet qui prévoit les méthodes nécessaires pour les mises à jour.

Si un objet o transmet sa variable d'instance v en paramètre à une méthode m, deux situations sont possibles :

si v est une variable primitive alors elle est passée par valeur : il est impossible de la modifier dans m pour que v en retour contienne cette nouvelle valeur.

si v est un objet alors m pourra modifier l'objet en utilisant une méthode de l'objet passé en paramètre.

3. L'émission de messages

Un message est émis lorsqu'on demande à un objet d'exécuter l'une de ses méthodes.

La syntaxe d'appel d'une méthode est : nom_objet.nom_méthode(parametre, ...) ;

Si la méthode appelée ne contient aucun paramètre, il faut laisser les parenthèses vides.

4. L'enchaînement de références à des variables et à des méthodes

Exemple : *System.out.println("bonjour");*

Deux classes sont impliquées dans l'instruction : System et PrintStream. La classe System possède une variable nommée out qui est un objet de type PrintStream. Println() est une méthode de la classe PrintStream. L'instruction signifie : « utilise la méthode println() de la variable out de la classe System ».

5. Les arguments variables (varargs)

Depuis Java 1.5, les varargs, spécifiés dans la JSR 201, permettent de passer un nombre non défini d'arguments d'un même type à une méthode. Ceci va éviter de devoir encapsuler ces données dans une collection.

Elle utilise un notation pour préciser la répétition d'un type d'argument utilisant trois petits points : ...

Exemple (java 1.5) :

```
public class TestVarargs {
public static void main(String[] args) { System.out.println("valeur
1 = " + additionner(1,2,3)); System.out.println("valeur 2 = " +
additionner(2,5,6,8,10));
}
```

```
public static int additionner(int ... valeurs) {
int total = 0;
for (int val : valeurs) {
total += val;
}
return total;
}
}
```

Résultat :

```
C:\tiger>java TestVarargs
valeur 1 = 6
valeur 2 = 31
```

L'utilisation de la notation ... permet le passage d'un nombre indéfini de paramètres du type précisé. Tous ces paramètres sont traités comme un tableau : il est d'ailleurs possible de fournir les valeurs sous la forme d'un tableau.

Exemple (java 1.5) :

```
public class TestVarargs2 {
public static void main(String[] args) {
int[] valeurs = {1,2,3,4};
System.out.println("valeur 1 = " + additionner(valeurs));
}
public static int additionner(int ... valeurs) {
int total = 0;
for (int val : valeurs) {
total += val;
}
return total;
}
}
```

Résultat :

```
C:\tiger>java TestVarargs2
valeur 1 = 10
```

Il n'est cependant pas possible de mixer des éléments unitaires et un tableau dans la liste des éléments fournis en paramètres.

Exemple (java 1.5) :

```
public class TestVarargs3 {
public static void main(String[] args) {
int[] valeurs = {1,2,3,4};
System.out.println("valeur 1 = " + additionner(5,6,7,valeurs));
}
public static int additionner(int ... valeurs) {
int total = 0;
for (int val : valeurs) {
total += val;
}
return total;
}
}
```

Résultat :

```
C:\tiger>javac −source 1.5 −target 1.5 TestVarargs3.java
TestVarargs3.java:7: additionner(int[]) in TestVarargs3 cannot be applied to (in
t,int,int,int[])
System.out.println("valeur 1 = " + additionner(5,6,7,valeurs));
^
1 error
```

6. La surcharge de méthodes

La surcharge d'une méthode permet de définir plusieurs fois une même méthode avec des arguments différents. Le compilateur choisi la méthode qui doit être appelée en fonction du nombre et du type des arguments. Ceci permet de simplifier l'interface des classes vis à vis des autres classes.

Une méthode est surchargée lorsqu'elle exécute des actions différentes selon le type et le nombre de paramètres transmis.

Il est donc possible de donner le même nom à deux méthodes différentes à condition que les signatures de ces deux méthodes soient différentes. La signature d'une méthode comprend le nom de la classe, le nom de la méthode et les types des paramètres.

Exemple :

```
class affiche{
public void afficheValeur(int i) {
System.out.println(" nombre entier = " + i);
}
public void afficheValeur(float f) {
System.out.println(" nombre flottant = " + f);
}
}
```

Il n'est pas possible d'avoir deux méthodes de même nom dont tous les paramètres sont identiques et dont seul le type retourné diffère.

Exemple :

```
class Affiche{
public float convert(int i){
return((float) i);
}
public double convert(int i){
return((double) i);
}
}
```

Résultat :

```
C:\>javac Affiche.java
Affiche.java:5: Methods can't be redefined with a different return type: double
convert(int) was float convert(int)
public double convert(int i){
^
1 error
```

7. Les constructeurs

La déclaration d'un objet est suivie d'une sorte d'initialisation par le moyen d'une méthode particulière appelée constructeur pour que les variables aient une valeur de départ. Elle n'est systématiquement invoquée que lors de la création d'un objet.

Le constructeur suit la définition des autres méthodes excepté que son nom doit obligatoirement correspondre à celui de la classe et qu'il n'est pas typé, pas même void, donc il ne peut pas y avoir d'instruction return dans un constructeur. On peut surcharger un constructeur.

La définition d'un constructeur est facultative. Si aucun constructeur n'est explicitement défini dans la classe, le compilateur va créer un constructeur par défaut sans argument. Dès qu'un constructeur est explicitement défini, le compilateur considère que le programmeur prend en charge la création des constructeurs et que le mécanisme par défaut, qui correspond à un constructeur sans paramètres, n'est pas mis en oeuvre. Si on souhaite maintenir ce mécanisme, il faut définir explicitement un constructeur sans paramètres en plus des autres constructeurs.

Il existe plusieurs manières de définir un constructeur :

le constructeur simple : ce type de constructeur ne nécessite pas de définition explicite : son existence découle automatiquement de la définition de la classe.

Exemple : *public MaClasse() {}*

2. le constructeur avec initialisation fixe : il permet de créer un constructeur par défaut

Exemple :

```
public MaClasse() {
nombre = 5;
}
```

le constructeur avec initialisation des variables : pour spécifier les valeurs de données à initialiser on peut les passer en paramètres au constructeur

Exemple :

```
public MaClasse(int valeur) {
nombre = valeur;
}
```

8. Les accesseurs

L'encapsulation permet de sécuriser l'accès aux données d'une classe. Ainsi, les données déclarées private à l'intérieur d'une classe ne peuvent être accédées et modifiées que par des méthodes définies dans la même classe. Si une autre classe veut accéder aux données de la classe, l'opération n'est possible que par l'intermédiaire d'une méthode de la classe prévue à cet effet. Ces appels de méthodes sont appelés « échanges de messages ».

Un accesseur est une méthode publique qui donne l'accès à une variable d'instance privée. Pour une variable d'instance, il peut ne pas y avoir d'accesseur, un seul accesseur en lecture ou un accesseur en lecture et un autre en écriture. Par convention, les accesseurs en lecture commencent par get et les accesseurs en écriture commencent par set.

Exemple :

```
private int valeur = 13;
public      int
getValeur(){
return(valeur
);
}
public void setValeur(int val) {
valeur = val;
}
```

Pour un attribut de type booléen, il est possible de faire commencer l'accesseur en lecture par is au lieu de get.

VI. L'héritage

L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Elle définit une relation entre deux classes :

une classe mère ou super−classe

une classe fille ou sous−classe qui hérite de sa classe mère

1. Le principe de l'héritage

Grâce à l'héritage, les objets d'une classe fille ont accès aux données et aux méthodes de la classe parente et peuvent les étendre. Les sous−classes peuvent redéfinir les variables et les méthodes héritées. Pour les variables, il suffit de les redéclarer sous le même nom avec un type différent. Les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge.

L'héritage successif de classes permet de définir une hiérarchie de classe qui se compose de super−classes et de sous−classes. Une classe qui hérite d'une autre est une sous−classe et celle dont elle hérite est une super−classe. Une classe peut avoir plusieurs sous−classes. Une classe ne peut avoir qu'une seule classe mère : il n'y a pas d'héritage multiple en Java.

Object est la classe parente de toutes les classes en Java. Toutes les variables et méthodes contenues dans Object sont accessibles à partir de n'importe quelle classe car par héritages successifs toutes les classes héritent d'Object.

2. La mise en oeuvre de l'héritage

On utilise le mot clé extends pour indiquer qu'une classe hérite d'une autre. En l'absence de ce mot réservé associé à une classe, le compilateur considère la classe Object comme classe mère.

Exemple : *class Fille extends Mere { ... }*

Pour invoquer une méthode d'une classe mère, il suffit d'indiquer la méthode préfixée par super. Pour appeler le constructeur de la classe mère, il suffit d'écrire super(paramètres) avec les paramètres adéquats.

Le lien entre une classe fille et une classe mère est géré par la plate−forme : une évolution des règles de gestion de la classe mère conduit à modifier automatiquement la classe fille dès que cette dernière est recompilée.

En Java, il est obligatoire dans un constructeur d'une classe fille de faire appel explicitement ou implicitement au constructeur de la classe mère.

3. L'accès aux propriétés héritées

Les variables et méthodes définies avec le modificateur d'accès public restent publiques à travers l'héritage et toutes les autres classes.

Une variable d'instance définie avec le modificateur private est bien héritée mais elle n'est pas accessible directement mais par les méthodes héritées.

Une variable définie avec le modificateur protected sera héritée dans toutes les classes filles qui pourront y accéder librement ainsi que les classes du même package.

4. La redéfinition d'une méthode héritée

La redéfinition d'une méthode héritée doit impérativement conserver la déclaration de la méthode parente (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identiques).

Si la signature de la méthode change, ce n'est plus une redéfinition mais une surcharge. Cette nouvelle méthode n'est pas héritée : la classe mère ne possède pas de méthode possédant cette signature.

5. Le polymorphisme

Le polymorphisme est la capacité, pour un même message de correspondre à plusieurs formes de traitements selon l'objet auquel ce message est adressé. La gestion du polymorphisme est assurée par la machine virtuelle dynamiquement à l'exécution.

6. Le transtypage induit par l'héritage facilite le polymorphisme

L'héritage définit un cast implicite de la classe fille vers la classe mère : on peut affecter à une référence d'une classe n'importe quel objet d'une de ses sous−classes.

Exemple : la classe Employe hérite de la classe Personne

```
Personne p = new Personne ("Dupond",
"Jean"); Employe e = new Employe("Durand",
"Julien", 10000);
p = e ; // ok : Employe est une sous−classe de
Personne Objet obj;
obj = e ; // ok : Employe hérite de Personne qui elle même hérite de Object
```

Il est possible d'écrire le code suivant si Employe hérite de Personne

Exemple :

Personne[] tab = new Personne[10];

tab[0] = new Personne("Dupond","Jean");

tab[1] = new Employe("Durand", "Julien", 10000);

Il est possible de surcharger une méthode héritée : la forme de la méthode à exécuter est choisie en fonction des paramètres associés à l'appel.

Compte tenu du principe de l'héritage, le temps d'exécution du programme et la taille du code source et de l'exécutable augmentent.

7. Les interfaces et l'héritage multiple

Avec l'héritage multiple, une classe peut hériter en même temps de plusieurs super−classes. Ce mécanisme n'existe pas en Java. Les interfaces permettent de mettre en oeuvre un mécanisme de remplacement.

Une interface est un ensemble de constantes et de déclarations de méthodes correspondant un peu à une classe abstraite. C'est une sorte de standard auquel une classe peut répondre. Tous les objets qui se conforment à cette interface (qui implémentent cette interface) possèdent les méthodes et les constantes déclarées dans celle−ci. Plusieurs interfaces peuvent être implémentées dans une même classe.

Les interfaces se déclarent avec le mot clé interface et sont intégrées aux autres classes avec le mot clé implements. Une interface est implicitement déclarée avec le modificateur abstract.

Déclaration d'une interface :

[public] interface nomInterface [extends nomInterface1, nomInterface2
...] { // insérer ici des méthodes ou des champs static
}

Implémentation d'une interface :

Modificateurs class nomClasse [extends superClasse] [implements nomInterface1, nomInterface 2, ...] {
//insérer ici des méthodes et des champs

}

Exemple :

```
interface AfficheType {
void afficherType();
}
class Personne implements AfficheType {
public void afficherType() {
System.out.println(" Je suis une personne ");
}
}
class Voiture implements AfficheType {
public void afficherType() {
System.out.println(" Je suis une voiture ");
}
}
```

Exemple : déclaration d'une interface à laquelle doit se conformer tout individus

```
interface Individu {
String getNom();
String getPrenom();
Date getDateNaiss();
}
```

Toutes les méthodes d'une interface sont abstraites : elles sont implicitement déclarées comme telles.

Une interface peut être d'accès public ou package. Si elle est publique, toutes ses méthodes sont implicitement publiques même si elles ne sont pas déclarées avec le modificateur public. Si elle est d'accès package, il s'agit d'une interface d'implémentation pour les autres classes du package et ses méthodes ont le même accès package : elles sont accessibles à toutes les classes du packages.

Les seules variables que l'on peut définir dans une interface sont des variables de classe qui doivent être constantes : elles sont donc implicitement déclarées avec le modificateur static et final même si elles sont définies avec d'autres modificateurs.

Exemple :

```
public interface MonInterface {
public int VALEUR=0;
void maMethode();
}
```

Toute classe qui implémente cette interface doit au moins posséder les méthodes qui sont déclarées dans l'interface.

L'interface ne fait que donner une liste de méthodes qui seront à définir dans les classes qui implémentent l'interface.

Les méthodes déclarées dans une interface publique sont implicitement publiques et elles sont héritées par toutes les classes qui implémentent cette interface. De telles classes doivent, pour être instanciables, définir toutes les méthodes héritées de l'interface.

Une classe peut implémenter une ou plusieurs interfaces tout en héritant de sa classe mère.

L'implémentation d'une interface définit un cast : l'implémentation d'une interface est une forme d'héritage. Comme pour l'héritage d'une classe, l'héritage d'une classe qui implémente une interface définit un cast implicite de la classe fille vers cette interface. Il est important de noter que dans ce cas il n'est possible de faire des appels qu'à des méthodes de l'interface. Pour utiliser des méthodes de l'objet, il faut définir un cast explicite : il est préférable de contrôler la classe de l'objet pour éviter une exception ClassCastException à l'exécution.

1. Les méthodes par défaut

Depuis les débuts de Java, il est possible d'utiliser l'héritage multiple avec les interfaces. Les méthodes par défaut de Java 8 permettent l'héritage multiple de comportement.

L'héritage d'interfaces est possible depuis la version 1.0 de Java. : une interface ne peut contenir que la déclaration de constantes et de méthodes mais elle ne contient pas leurs traitements. C'est l'héritage multiple de type (multiple inheritance of type).

Avec les méthodes par défaut de Java 8, Java introduit la possibilité d'héritage multiple de comportement (multiple inheritance of behaviour) mais ne permet toujours pas l'héritage multiple d'état (multiple inheritance of state) puisque seules les interfaces sont concernées par l'héritage multiple.
Les interfaces sont couplées avec les classes qui les implémentent : par exemple, il n'est pas possible d'ajouter une méthode à une interface sans devoir modifier les classes qui l'implémentent directement.
Avant Java 8, la modification d'une ou plusieurs méthodes d'une interface oblige à adapter en conséquence toutes les classes qui l'implémentent. La seule solution pour éviter cela aurait été de créer une nouvelle version de l'API et les deux versions auraient dues cohabiter, ce qui aurait impliqué des problèmes pour maintenir et utiliser l'API.
L'ajout des lambdas dans certaines classes de base du JDK, notamment le package java.util, aurait eu beaucoup d'impacts sans les méthodes par défaut. L'intérêt initial des méthodes par défaut est donc de maintenir la compatibilité ascendante des API.

A partir de Java 8, il est possible d'utiliser les méthodes par défaut (default method) dans une interface. Elles permettent de définir le comportement d'une méthode dans l'interface dans laquelle elle est définie. Si aucune implémentation de la méthode n'est fournie dans une classe qui implémente l'interface alors c'est le comportement défini dans l'interface qui sera utilisé.
Une méthode par défaut est déclarée en utilisant le mot clé default. Le corps de la méthode contient l'implémentation des traitements.

Exemple (code Java 8) :

```
public interface MonInterface {
default void maMethode() {
System.out.println("Implementation par defaut");
}
}
```

Les méthodes par défaut devraient surtout être utilisées pour maintenir une compatibilité ascendante afin de permettre l'ajout d'une méthode à une interface existante sans avoir à modifier les classes qui l'implémentent.

Les méthodes par défaut peuvent aussi éviter d'avoir à implémenter une classe abstraite qui contient les traitements par défaut de méthodes héritées dans les classes filles concrètes. Ces implémentations peuvent directement être codées dans des méthodes par défaut de l'interface. Les méthodes par défaut ne remplacent cependant pas complètement les classes abstraites qui peuvent avoir des constructeurs et des membres sous la forme de variables d'instance ou de classe.

Remarque : bien qu'il soit possible en Java 8 de définir des méthodes static et par défaut dans une interface, ce n'est pas possible dans la définition du type d'une annotation.

L'héritage multiple de comportement permet par exemple que de mêmes méthodes par défaut, avec des signatures identiques, soient définies dans plusieurs interfaces héritées par une interface fille. Il est probable que chaque implémentation de ces méthodes soit différente : le compilateur a besoin de règles pour déterminer quelle implémentation il doit utiliser :

la redéfinition d'une méthode par une classe ou une super−classe est toujours prioritaire par rapport à une méthode par défaut

l'implémentation choisie est celle par défaut de l'interface la plus spécifique

Ainsi une interface peut être modifiée en ajoutant une méthode sans compromettre sa compatibilité ascendante sous réserve qu'elle implémente cette méthode en tant que méthode par défaut.

Les méthodes par défaut sont virtuelles comme toutes les autres méthodes mais elles proposent une implémentation par défaut qui sera invoquée si la classe implémentant l'interface ne redéfinit pas explicitement la méthode. Une classe qui implémente une interface n'a donc pas l'obligation de redéfinir une méthode par défaut. Si celle−ci n'est pas redéfinie alors c'est l'implémentation contenue dans l'interface qui est utilisée.

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public interface Service {
default void afficherNom() {
System.out.println("Nom du service : inconnu");
}
}
```

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
```

```
public class MonService implements Service {
}
```

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public class TestMethodeParDefaut {
public static void main(String[] args) {
Service service = new MonService(){};
service.afficherNom();
}
}
```

Résultat :

Nom du service : inconnu

Si la classe redéfinit la méthode alors c'est l'implémentation de la méthode qui est utilisée.

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public class MonService implements Service {
@Override
public void afficherNom() {
System.out.println("Nom du service : mon service");
}
}
```

Résultat :

Nom du service : mon service

Il est possible d'utiliser l'annotation @Override sur la méthode redéfinie qu'elle soit par défaut ou non.

Il est aussi possible de créer directement une instance d'une interface si toutes ses méthodes sont des méthodes par défaut.

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public class TestMethodeParDefaut {
public static void main(String[] args) {
Service service = new Service(){};
service.afficherNom();
}
}
```

Résultat :

Nom du service : inconnu

Une interface peut hériter d'une autre interface qui contient une méthode par défaut et peut redéfinir cette méthode.

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public interface ServiceSpecial extends Service {
default void afficherNom() {
System.out.println("Nom du service special : inconnu");
}
}
```

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public class MonService implements ServiceSpecial {
}
```

Résultat de l'exécution de la classe TestMethodeParDefaut :

Nom du service special : inconnu

L'interface fille peut redéfinir la méthode sans la déclarer par défaut : dans ce cas, elle est redéfinie comme étant abstraite.

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public interface ServiceSpecial extends Service {
void afficherNom();
}
```

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public class MonService implements ServiceSpecial {
}
```

Résultat :

```
C:\java\TestJava8\src>javac −cp .
tg/epl/cours/test/java8/MonService.java
tg/epl/cours\test\java8\MonService.java:3: error: MonService is not
abstract an d does not override abstract method afficherNom() in
ServiceSpecial public class MonService implements ServiceSpecial {
^
1 error
```

Une classe peut implémenter deux interfaces qui définissent la méthode par défaut avec des implémentations par défaut différentes.

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public interface Groupe {
default void afficherNom() {
```

System.out.println("Nom du groupe : inconnu");
}
}

Exemple (code Java 8) :
package tg.poo.cours.epl.ul;
public class MonService implements Service, Groupe {
}

Dans ce cas, le compilateur lève une erreur qui précise le nom de la méthode par défaut et les interfaces concernées car il ne peut pas décider quelle implémentation il doit utiliser.

Résultat :
C:\java\TestJava8\src>javac −cp .
tg/epl/cours/test/java8/MonService.java
tg/epl/cours\test\java8\MonService.java:3: error: class MonService inherits unr elated defaults for afficherNom() from types Service and Groupe public class MonService implements Service, Groupe {
^
1 error

Pour régler le problème, la classe doit explicitement redéfinir la méthode.

Exemple (code Java 8) :
package tg.poo.cours.epl.ul;
public class MonService implements Service, Groupe {
@Override
public void afficherNom() {
System.out.println("Nom du service : mon service");
}
}

La redéfinition de la méthode dans la classe peut explicitement invoquer la méthode par défaut d'une des interfaces en utilisant la syntaxe : nom du type de l'interface, point, super, point le nom de la méthode.

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public class MonService implements Service, Groupe {
@Override
public void afficherNom() {
Groupe.super.afficherNom();
}
}
```

Résultat de l'exécution de la classe TestMethodeParDefaut :
Nom du groupe : inconnu

Il est possible que deux interfaces définissent la même méthode par défaut avec des implémentations différentes.
Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public interface Service {
default void afficherNom() {
System.out.println("Nom du service : inconnu");
}
}
```

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public interface ServiceEtendu {
default void afficherNom() {
System.out.println("Nom du service etendu : inconnu");
}
```

}

Une troisième interface peut hériter de ces deux interfaces.

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public interface ServiceSpecial extends Service, ServiceEtendu {
}
```

Le compilateur génère une erreur car il n'est pas en mesure de déterminer laquelle des deux implémentations il doit utiliser.

Résultat :

```
C:\java\TestJava8\src>javac −cp .
tg/epl/cours/test/java8/ServiceSpecial.java
tg/epl/cours\test\java8\ServiceSpecial.java:3: error: interface
ServiceSpecial inherits unrelated defaults for affichерNom() from types
Service and ServiceEtte ndu
public interface ServiceSpecial extends Service, ServiceEtendu {
^
1 error
```

Une classe peut implémenter deux interfaces :

- une qui définit une méthode par défaut
- une autre qui définit la même méthode mais abstraite

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public interface Groupe {
void afficherNom();
```

}

Exemple (code Java 8) :

package tg.poo.cours.epl.ul;

public class MonService implements Service, Groupe {

}

Dans ce cas, le compilateur lève une erreur car il ne peut pas décider de prendre la méthode par défaut.

Résultat :

C:\java\TestJava8\src>javac −cp .

tg/epl/cours/test/java8/MonService.java

tg/epl/cours\test\java8\MonService.java:4: error: MonService is not abstract an d does not override abstract method afficherNom() in Groupe

public class MonService implements Service, Groupe {

^

1 error

Pour régler le problème, la classe doit explicitement redéfinir la méthode, éventuellement en invoquant la méthode par défaut de l'interface.

Si aucune des interfaces ne propose de méthodes par défaut, alors il n'y a pas d'ambiguïté et cette situation est celle qui pouvait exister avant Java 8. Une classe qui implémente ces interfaces doit fournir une implémentation pour chacune des méthodes ou être déclarée abstraite.

Il est possible qu'une classe hérite d'une classe et implémente une interface avec une méthode par défaut qui est implémentée par la classe mère.

Exemple (code Java 8) :

package tg.poo.cours.epl.ul;

public class MonService implements ServiceEtendu {

}

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public class ServiceComptable extends MonService implements Service{
}
```

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;

public class TestMethodeParDefaut {
public static void main(String[] args) {
Service service = new ServiceComptable();
service.afficherNom();
}
}
```

Résultat :

Nom du service etendu : inconnu

Dans ce cas, c'est la classe mère qui prévaut et la méthode par défaut de l'interface Service est ignorée par le compilateur. En application de la règle «l'implémentation d'une classe ou super−classe est prioritaire», c'est la méthode getName() héritée de la classe MonService qui est utilisée.

Les méthodes par défaut sont virtuelles comme toutes les méthodes en Java.

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public interface ServiceEtendu extends Service {
@Override
default void afficherNom() {
System.out.println("Nom du service etendu : inconnu");
}
}
```

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public interface ServiceDedie extends Service {
}
```

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public class MonService implements ServiceDedie, ServiceEtendu {
}
```

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;

public class TestMethodeParDefaut {
public static void main(String[] args) {
MonService monService = new MonService();
monService.afficherNom();
Service service = new MonService();
service.afficherNom();
}
}
```

Résultat :

```
Nom du service etendu : inconnu
Nom du service etendu : inconnu
```

Comme la méthode par défaut n'est pas redéfinie, le compilateur applique les règles pour déterminer l'implémentation de la méthode par défaut à utiliser : dans le cas ci−dessus, c'est celle de l'interface ServiceEtendu qui est la plus spécifique car elle redéfinit la méthode héritée de l'interface Service. Peu importe le type de la variable, c'est l'implémentation de l'instance créée qui est utilisée.

La règle qui veut qu'une implémentation d'une classe prévale toujours sur une méthode par défaut implique plusieurs choses :

elle assure la compatibilité avec les classes antérieures à Java 8 : l'ajout de méthodes par défaut n'a pas d'effet sur du code qui fonctionnait avant l'ajout des méthodes par défaut

elle empêche de définir de manière utile les méthodes de la classe Object. Le compilateur lève une erreur si une méthode ayant la signature d'une des méthodes toString(), equals() ou hashCode() de la classe Object est définie comme étant par défaut dans une interface

Exemple (code Java 8) :

```
package tg.poo.cours.epl.ul;
public interface Service {
default String toString() {
return "";
}
default boolean equals(Object o) {
return false;
}
default public int hashCode() {
return 0;
}
}
```

Résultat :

```
C:\java\TestJava8\src>jav
ac −cp . tg/epl/cours/test/java8/Service.java
tg/epl/cours\test\java8\Service.java:5: error: default method toString in
inter face Service overrides a member of java.lang.Object
default String toString() {
^
tg/epl/cours\test\java8\Service.java:9: error: default method equals in
interfa ce Service overrides a member of java.lang.Object
default boolean equals(Object
o) {
```

^

tg/epl/cours\test\java8\Service.java:13: error: default method hashCode in

inte rface Service overrides a member of java.lang.Object

default public int hashCode() {

^

3 errors

2. Les interfaces locales

Jusqu'à Java 15, il n'est pas possible de définir des interfaces locales.

Exemple (code Java 15) :

```
public class TestInterfaceLocal {
public void traiter() {
interface MonInterface {};
}
}
```

Résultat :

```
C:\java>javac −version
javac 15
C:\java>javac TestInterfaceLocal.java
TestInterfaceLocal.java:5: error: interface not
allowed here
interface MonInterface {};
^
        error C:\java>
```

Java 16 permet de définir des interfaces locales, qui ne pourront donc être utilisées que dans la classe où elles sont définies.

Exemple (code Java 16) :

```
public class InterfaceLocale {
```

```
public void traiter() {
interface MonInterface {
public default void afficher() {
System.out.println("Hello");
}
};
(new MonInterface() {}).afficher();
}
}
```

Les interfaces locales ne peuvent pas capturer les variables du contexte englobant comme les paramètres de la méthode par exemple.

Exemple (code Java 16) :

```
public class InterfaceLocale {
public void traiter(int valeur) {
interface MonInterface {
public default void afficher() {
System.out.println(valeur);
}
};
(new MonInterface() {}).afficher();
}
}
```

Résultat :

```
C:\java>javac InterfaceLocale.java
InterfaceLocale.java:8: error: non−static variable valeur cannot be referenced from a
static context
System.out.println(valeur);
^
        error C:\java>
```

Les interfaces locales peuvent capturer les variables static du contexte englobant.

Exemple (code Java 16) :

```
public class InterfaceLocale {
static int valeur = 10;
public void traiter() {
interface MonInterface {
public default void afficher() {
System.out.println(valeur);
}
};
(new MonInterface() {}).afficher();
}
}
```

L'héritage de méthodes statiques

Toutes les méthodes, incluant les méthodes statiques, sont héritées d'une super−classe du moment qu'elles soient accessibles par la classe fille.

Exemple :

```
package tg.epl.cours;
public class MaClasseMere {
public static void maMethode() {
System.out.println("MaClasseMere");
}
}
```

Exemple :

```
package tg.epl.cours;
public class MaClasseFille extends MaClasseMere {
}
```

Exemple :

```
package tg.epl.cours;
public class TestHeritageStatic {
```

```
public static void main(String[] args) {
MaClasseMere.maMethode();
MaClasseFille.maMethode();
}
}
```

Résultat :
MaClasseMere
MaClasseMere

Dans le cas des méthodes statiques, il y a cependant une restriction qui interdit de redéfinir une méthode statique héritée.
Pourtant, il est possible d'écrire :

Exemple :

```
package tg.epl.cours;
public class MaClasseFille extends MaClasseMere {
public static void maMethode() {
System.out.println("MaClasseFille");
}
}
```

Résultat de l'exécution de la méthode TestHeritageStatic :
MaClasseMere
MaClasseFille

Une méthode static ne peut pas être redéfinie (overriden) mais il est possible de définir une méthode dans la classe fille avec la même signature. Si une méthode statique définie dans une classe mère est définie de manière identique dans une classe fille, celle−ci n'est pas une redéfinition mais elle masque (hidden) la méthode de la classe mère.

Exemple :

```
package tg.epl.cours;
public class MaClasseMere {
public static void maMethode() {
System.out.println("MaClasseMere");
}
public static void monAutreMethode() {
maMethode();
}
}
```

Exemple :

```
package tg.epl.cours;
public class MaClasseFille extends MaClasseMere {
public static void maMethode() {
System.out.println("MaClasseFille");
}
}
```

Exemple :

```
package tg.epl.cours;
public class TestHeritageStatic {
public static void main(String[] args) {
MaClasseMere.maMethode();
MaClasseMere.monAutreMethode();
MaClasseFille.maMethode();
MaClasseFille.monAutreMethode();
}
}
```

Résultat :

```
MaClasseMere
MaClasseMere
MaClasseFille
MaClasseMere
```

Il n'est donc pas possible d'utiliser l'annotation @Override.

Exemple (code Java 6) :

```
package tg.epl.cours;
public class MaClasseFille extends MaClasseMere {
@Override
public static void maMethode() {
System.out.println("MaClasseFille");
}
}
```

Résultat :

```
C:\java\Test\src\tg/epl/cours\dej>javac
MaClasseFille.java
MaClasseFille.java:3:
error: cannot find symbol
public class MaClasseFille extends MaClasseMere {
^
symbol: class MaClasseMere
MaClasseFille.java:5:
error: method does not override or implement a method from a
supertype @Override
^
2 errors
```

Il n'est pas possible de redéfinir une méthode statique dans une classe fille si cette redéfinition n'est pas statique.

Exemple :

```
package tg.epl.cours;
public class MaClasseFille extends MaClasseMere {
```

```
public void maMethode() {
System.out.println("MaClasseFille");
}
}
```

Résultat :

```
C:\Users\jm\workspace\Test\src>javac
tg/epl/cours/java/MaClasseFille.java
tg/epl/cours\java\MaClasseFille.java:5:
error: maMethode() in MaClasseFille cannot override maMethode() in
MaClasseMere
public                void
maMethode() {
^
overridden method is static
1 error
```

La redéfinition des méthodes d'instances implique une résolution à l'exécution. Les méthodes statiques sont des méthodes de classes : leurs résolutions sont toujours faites par le compilateur à la compilation. Il n'est donc pas possible d'utiliser le polymorphisme sur des méthodes statiques, celles−ci étant résolues par le compilateur.

Exemple :

```
package tg.epl.cours;
public class Test {
public static void main(String[] args) {
MaClasseMere mere = new MaClasseMere();
mere.maMethode();
MaClasseFille fille = new MaClasseFille();
fille.maMethode();
mere = new MaClasseFille();
mere.maMethode();
}
```

}

Résultat :
MaClasseMere
MaClasseFille
MaClasseMere

Ce comportement est dû au fait que la méthode n'est pas redéfinie mais masquée. Les accès aux méthodes statiques sont toujours résolus à la compilation : le compilateur utilise le type de la variable et pas le type de l'instance qui invoque la méthode. L'invocation d'une méthode statique à partir d'une instance est possible en Java mais le compilateur émet un avertissement pour préconiser l'utilisation de la classe pour invoquer la méthode et ainsi éviter toute confusion sur la méthode invoquée.
Le compilateur remplace l'instance par la classe de son type, ce qui permet à l'exemple ci−dessous de se compiler et de s'exécuter correctement.
Exemple :

```
package tg.epl.cours;
public class Test {
public static void main(String[] args) {
MaClasseMere mere = null;
mere.maMethode();
}
}
```

Résultat :
MaClasseMere

La redéfinition (overriding) est une fonctionnalité offerte par les langages de POO qui permet de mettre en ouvre une forme de polymorphisme. Une sous−classe fournit une implémentation dédiée d'une méthode héritée de sa super−classe : les signatures des deux méthodes doivent être les mêmes. Le choix de la méthode à exécuter est déterminé à l'exécution en fonction du type de l'objet qui l'invoque.

La surcharge (overload) est une fonctionnalité offerte par les langages de POO qui permet de mettre en oeuvre une forme de polymorphisme. Elle permet de définir différentes méthodes ayant le même nom avec le nombre et/ou le type des paramètres différent.
Le choix de la méthode à exécuter est déterminée statiquement par le compilateur en fonction des paramètres utilisés à l'invocation.

3. Des conseils sur l'héritage

Lors de la création d'une classe « mère » il faut tenir compte des points suivants :

la définition des accès aux variables d'instances, très souvent privées, doit être réfléchie entre protected et private

pour empêcher la redéfinition d'une méthode ou sa surcharge, il faut la déclarer avec le modificateur final

Lors de la création d'une classe fille, pour chaque méthode héritée qui n'est pas final, il faut envisager les cas suivants :

la méthode héritée convient à la classe fille : on ne doit pas la redéfinir

la méthode héritée convient mais partiellement du fait de la spécialisation apportée par la classe fille : il faut la redéfinir voire la surcharger. La plupart du temps une redéfinition commencera par appeler la méthode héritée (en utilisant le mot clé super) pour garantir l'évolution du code

la méthode héritée ne convient pas : il faut redéfinir ou surcharger la méthode sans appeler la méthode héritée lors de la redéfinition.

VII. Les packages

En Java, il existe un moyen de regrouper des classes voisines ou qui couvrent un même domaine : ce sont les packages.

1. La définition d'un package

Pour réaliser un package, on écrit un nombre quelconque de classes dans plusieurs fichiers d'un même répertoire et au début de chaque fichier on met la directive ci−dessous où nom−du−package doit être composé des répertoires séparés par un caractère point :

```
package nom−du−package;
```

La hiérarchie d'un package se retrouve dans l'arborescence du disque dur puisque chaque package est dans un répertoire nommé du nom du package.

D'une façon générale, l'instruction package associe toutes les classes qui sont définies dans un fichier source à un même package.
Le mot clé package doit être la première instruction dans un fichier source et il ne doit être présent qu'une seule fois dans le fichier source (une classe ne peut pas appartenir à plusieurs packages).

2. Les importations

Pour pouvoir utiliser un type en Java, il faut utiliser le nom pleinement qualifié du type qui inclue le nom du package avec la notation utilisant un point.
Afin de réduire la verbosité lors de l'utilisation de type, il est possible d'utiliser les imports. Ils utilisent le mot clé import qui définit un alias du nom pleinement qualifié d'un type vers simplement le nom du type.
Il y a deux manière d'utiliser les imports :

- préciser un nom de classe ou d'interface qui sera l'unique entité importée
- ou indiquer un package suivi d'un point et d'un caractère * indiquant toutes les classes et interfaces définies dans le package

Exemple	Rôle
import nomPackage.*;	toutes les classes du package sont importées
import nomPackage.nomClasse;	appel à une seule classe : l'avantage de cette notation est de réduire le temps de compilation

Attention : l'astérisque n'importe pas les sous−packages. Par exemple, il n'est pas possible d'écrire import java.*.

L'utilisation de joker dans les imports est un choix :

- ne pas les utiliser permet de déterminer précisément les classes utilisées
- l'utilisation de joker permet de réduire le nombre d'import

Généralement, les IDE proposent une fonctionnalité qui permet de réorganiser les imports en

- remplaçant les jokers par les classes utilisées
- triant les classes utilisées par ordre alphabétique

Les imports sont traités par le compilateur : le bytecode généré sera le même qu'une clause import utilise le nom pleinement qualifié d'une classe ou un nom comportant un joker.
Le bytecode ne contient pas les imports mais simplement le nom pleinement qualifiés des classes.
L'utilisation d'un joker dans un import n'a aucun impact sur les performances ou la consommation mémoire à l'exécution.
L'utilisation de joker dans les imports peut cependant induire une collision de classes si deux classes ayant le même nom dans deux packages différents sont importés en utilisant un joker.
L'utilisation de joker peut aussi induire de futurs problèmes de compilation si deux import utilisent un joker et qu'une classe ayant un nom existant dans un des packages est ajoutée dans l'autre package ultérieurement. La classe ne se compilera alors plus.
En précisant son nom complet, il est possible d'appeler une méthode d'une classe sans utiliser son importation :
nomPackage.nomClasse.nomméthode(arg1, arg2 ...)
Il existe plusieurs types de packages : le package par défaut (identifié par le point qui représente le répertoire courant et permet de localiser les classes qui ne sont pas associées à un package particulier), les packages standard qui sont empaquetés dans le fichier classes.zip (Java 1.0 et 1.1) et rt.jar (à partir de Java 1.2) et les packages personnels.
Le compilateur implémente automatiquement une instruction import lors de la compilation d'un programme Java même si elle ne figure pas explicitement au début du programme : import java.lang.*;. Ce package contient entre autres les classes de base de tous les objets Java dont la classe Object.
Un package par défaut est systématiquement attribué par le compilateur aux classes qui sont définies sans déclarer explicitement une appartenance à un package. Ce package par défaut correspond au répertoire courant qui est le répertoire de travail.

3. Les importations statiques

Jusqu'à la version 1.4 de Java, pour utiliser un membre statique d'une classe, il fallait obligatoirement préfixer ce membre par le nom de la classe qui le contient.
Par exemple, pour utiliser la constante Pi définie dans la classe java.lang.Math, il est nécessaire d'utiliser Math.PI

Exemple :

```
public class TestStaticImportOld {
```

```
public static void main(String[] args) {
System.out.println(Math.PI);
System.out.println(Math.sin(0));
}
}
```

Java 1.5 propose une solution pour réduire le code à écrire concernant les membres statiques en proposant une nouvelle fonctionnalité concernant l'importation de package : l'import statique (static import).
Ce nouveau concept permet d'appliquer les mêmes règles aux membres statiques qu'aux classes et interfaces pour l'importation classique.
Cette nouvelle fonctionnalité est développée dans la JSR 201. Elle s'utilise comme une importation classique en ajoutant le mot clé static.

Exemple (java 1.5) :

```
import static java.lang.Math.*;
public class TestStaticImport {
public static void main(String[] args) {
System.out.println(PI);
System.out.println(sin(0));
}
}
```

L'utilisation de l'importation statique s'applique à tous les membres statiques : constantes et méthodes statiques de l'élément importé.La collision de classes
Deux classes entrent en collision lorsqu'elles portent le même nom mais qu'elles sont définies dans des packages différents. Dans ce cas, il faut qualifier explicitement le nom de la classe avec le nom complet du package.

4. Les packages et l'environnement système

Les classes Java sont chargées par le compilateur (au moment de la compilation) et par la machine virtuelle (au moment de l'exécution). Les techniques de chargement des classes

varient en fonction de l'implémentation de la machine virtuelle. Dans la plupart des cas, une variable d'environnement CLASSPATH référence tous les répertoires qui hébergent des packages susceptibles d'être importés.

Exemple sous Windows :

CLASSPATH = .;C:\MonApplication\lib\classes.zip;C:\MonApplication\classes

L'importation des packages ne fonctionne que si le chemin de recherche spécifié dans une variable particulière pointe sur les packages, sinon le nom du package devra refléter la structure du répertoire où il se trouve. Pour déterminer l'endroit où se trouvent les fichiers .class à importer, le compilateur utilise une variable d'environnement dénommée CLASSPATH. Le compilateur peut lire les fichiers .class comme des fichiers indépendants ou comme des fichiers ZIP ou JAR dans lesquels les classes sont réunies et compressées.

VIII. Les classes internes

Les classes internes (inner classes) sont une extension du langage Java introduite dans la version 1.1 de Java. Ce sont des classes qui sont définies dans une autre classe. Les difficultés dans leur utilisation concernent leur visibilité et leur accès aux membres de la classe dans laquelle elles sont définies.

Exemple très simple :

```
public class ClassePrincipale1 {
class ClasseInterne {
}
}
```

Les classes internes sont particulièrement utiles pour :

- permettre de définir une classe à l'endroit où une seule autre en a besoin
- définir des classes de type adapter (essentiellement à partir du JDK 1.1 pour traiter des événements émis par les interfaces graphiques)
- définir des méthodes de type callback d'une façon générale

Pour permettre de garder une compatibilité avec la version précédente de la JVM, seul le compilateur a été modifié. Le compilateur interprète la syntaxe des classes internes pour modifier le code source et générer du bytecode compatible avec la première JVM.

Il est possible d'imbriquer plusieurs classes internes. Java ne possède pas de restrictions sur le nombre de classes qu'il est ainsi possible d'imbriquer. En revanche une limitation peut intervenir au niveau du système d'exploitation en ce qui concerne la longueur du nom du fichier .class généré pour les différentes classes internes.
Si plusieurs classes internes sont imbriquées, il n'est pas possible d'utiliser un nom pour la classe qui soit déjà attribué à une de ses classes englobantes. Le compilateur génèrera une erreur à la compilation.

Exemple :

```
public class ClassePrincipale6 {
class ClasseInterne1 {
class ClasseInterne2 {
class ClasseInterne3 {
}
}
}
}
```

Le nom de la classe interne utilise la notation qualifiée avec le point préfixé par le nom de la classe principale. Ainsi, pour utiliser ou accéder à une classe interne dans le code, il faut la préfixer par le nom de la classe principale suivi d'un point.

Cependant cette notation ne représente pas physiquement le nom du fichier qui contient le bytecode. Le nom du fichier qui contient le bytecode de la classe interne est modifié par le compilateur pour éviter des conflits avec d'autres noms d'entités : à partir de la classe principale, le point de séparation entre chaque classe interne est remplacé par un caractère $ (dollar).

Par exemple, la compilation du code de l'exemple précédent génère quatre fichiers contenant le bytecode :
ClassePrincipale6$ClasseInterne1$ClasseInterne2$ClasseInterne3.class
ClassePrincipale6$ClasseInterne1$ClasseInterne2.class
ClassePrincipale6$ClasseInterne1.class

ClassePrincipale6.class

L'utilisation du signe $ entre la classe principale et la classe interne permet d'éviter des confusions de nom entre le nom d'une classe appartenant à un package et le nom d'une classe interne.

L'avantage de cette notation est de créer un nouvel espace de nommage qui dépend de la classe et pas d'un package. Ceci renforce le lien entre la classe interne et sa classe englobante.

C'est le nom du fichier qu'il faut préciser lorsque l'on tente de charger la classe avec la méthode forName() de la classe Class. C'est aussi sous cette forme qu'est restitué le résultat d'un appel aux méthodes getClass().getName() sur un objet qui est une classe interne.

Exemple :

```
public class ClassePrincipale8 {
public class ClasseInterne {
}
public static void main(String[] args) { ClassePrincipale8
cp = new ClassePrincipale8();
ClassePrincipale8.ClasseInterne ci = cp. new
ClasseInterne() ;
System.out.println(ci.getClass().getName());
}
}
```

Résultat :

```
java ClassePrincipale8
ClassePrincipale8$ClasseInterne
```

L'accessibilité à la classe interne respecte les règles de visibilité du langage. Il est même possible de définir une classe interne private pour limiter son accès à sa seule classe principale.

Exemple :

```
public class ClassePrincipale7 {
private class ClasseInterne {
}
```

}

Il n'est pas possible de déclarer des membres statiques dans une classe interne :
Exemple :

```
public class ClassePrincipale10 {
public class ClasseInterne {
static int var = 3;
}
}
```

Résultat :

```
javac ClassePrincipale10.java
ClassePrincipale10.java:3: Variable var can't be static in inner class
ClassePri ncipale10. ClasseInterne. Only members of interfaces and
top−level classes can be static.
static int var = 3;
^
1 error
```

Pour pouvoir utiliser une variable de classe dans une classe interne, il faut la déclarer dans sa classe englobante.
Il existe quatre types de classes internes :

- les classes internes non statiques : elles sont membres à part entière de la classe qui les englobe et peuvent accéder à tous les membres de cette dernière
- les classes internes locales : elles sont définies dans un bloc de code. Elles peuvent être static ou non.
- les classes internes anonymes : elles sont définies et instanciées à la volée sans posséder de nom
- les classe internes statiques : elles sont membres à part entière de la classe qui les englobe et peuvent accéder uniquement aux membres statiques de cette dernière

1. Les classes internes non statiques

Les classes internes non statiques (member inner−classes) sont définies dans une classe dite « principale » (top−level class) en tant que membres de cette classe. Leur avantage est de pouvoir accéder aux autres membres de la classe principale même ceux déclarés avec le modificateur private.

Exemple :

```
public class ClassePrincipale20 {
private int valeur = 1;
class ClasseInterne {
public void afficherValeur() {
System.out.println("valeur = "+valeur);
}
}
public static void main(String[] args) {
ClassePrincipale20 cp = new
ClassePrincipale20(); ClasseInterne ci = cp.
new ClasseInterne(); ci.afficherValeur();
}
}
```

Résultat :

```
C:\testinterne>javac ClassePrincipale20.java
C:\testinterne>java ClassePrincipale20
valeur = 1
```

Le mot clé this fait toujours référence à l'instance en cours. Ainsi this.var fait référence à la variable var de l'instance courante. L'utilisation du mot clé this dans une classe interne fait donc référence à l'instance courante de cette classe interne.

Exemple :

```
public class ClassePrincipale16 {
class ClasseInterne {
int var = 3;
```

```
public void affiche() {
System.out.println("var                = "+var);
System.out.println("this.var = "+this.var);
}
}
ClasseInterne ci = this. new ClasseInterne();
public static void main(String[] args) {
ClassePrincipale16 cp = new
ClassePrincipale16(); ClasseInterne ci = cp.
new ClasseInterne(); ci.affiche();
}
}
```

Résultat :

```
C:\>java ClassePrincipale16

 var    = 3

this.var = 3
```

Une classe interne a accès à tous les membres de sa classe principale. Dans le code, pour pouvoir faire référence à un membre de la classe principale, il suffit simplement d'utiliser son nom de variable.

Exemple :

```
public class ClassePrincipale17 {
int valeur = 5;
class ClasseInterne {
int var = 3;
public void affiche() {
System.out.println("var                = "+var);
System.out.println("this.var = "+this.var);
System.out.println("valeur             = "+valeur);
```

```
}
}
ClasseInterne ci = this. new ClasseInterne();
public static void main(String[] args) {
ClassePrincipale17 cp = new
ClassePrincipale17(); ClasseInterne ci = cp.
new ClasseInterne(); ci.affiche();
}
}
```

Résultat :

```
C:\testinterne>java ClassePrincipale17

 var       = 3
 this.var  = 3
 valeur    = 5
```

La situation se complique un peu plus si la classe principale et la classe interne possèdent toutes les deux un membre de même nom. Dans ce cas, il faut utiliser la version qualifiée du mot clé this pour accéder au membre de la classe principale. La qualification se fait avec le nom de la classe principale ou plus généralement avec le nom qualifié d'une des classes englobantes.

Exemple :

```
public class ClassePrincipale18 {
int var = 5;
class ClasseInterne {
int var = 3;
public void affiche() {
System.out.println("var                              = "+var);
System.out.println("this.var                         = "+this.var);
System.out.println("ClassePrincipale18.this.var      = "
+ClassePrincipale18.this.var);
}
```

```
}
ClasseInterne ci = this. new ClasseInterne();
public static void main(String[] args) {
ClassePrincipale18 cp = new ClassePrincipale18();
ClasseInterne ci = cp. new ClasseInterne();
ci.affiche();
}
}
```

Résultat :

```
C:\>java
ClassePrincipale18
var                          = 3
this.var                     = 3
ClassePrincipale18.this.var = 5
```

Comme une classe interne ne peut être nommée du même nom que l'une de ses classes englobantes, ce nom qualifié est unique et il ne risque pas d'y avoir de confusion.
Le nom qualifié d'une classe interne est nom_classe_principale.nom_classe_interne. C'est donc le même principe que celui utilisé pour qualifier une classe contenue dans un package. La notation avec le point est donc légèrement étendue.
L'accès aux membres de la classe principale est possible car le compilateur modifie le code de la classe principale et celui de la classe interne pour fournir à la classe interne une référence sur la classe principale.
Le code de la classe interne est modifié pour :

- ajouter une variable privée finale du type de la classe principale nommée this$0
- ajouter un paramètre supplémentaire dans le constructeur qui sera la classe principale et qui va initialiser la variable this$0
- utiliser cette variable pour préfixer les attributs de la classe principale utilisés dans la classe interne.

La code de la classe principale est modifié pour :

ajouter une méthode static pour chaque champ de la classe principale qui attend en paramètre un objet de la classe principale. Cette méthode renvoie simplement la valeur du champ. Le nom de cette méthode est de la forme access$0

modifier le code d'instanciation de la classe interne pour appeler le constructeur modifié

Dans le bytecode généré, une variable privée finale contient une référence vers la classe principale. Cette variable est nommée this$0. Comme elle est générée par le compilateur, cette variable n'est pas utilisable dans le code source. C'est à partir de cette référence que le compilateur peut modifier le code pour accéder aux membres de la classe principale.

Pour pouvoir avoir accès aux membres de la classe principale, le compilateur génère dans la classe principale des accesseurs sur ses membres. Ainsi, dans la classe interne, pour accéder à un membre de la classe principale, le compilateur appelle un de ses accesseurs en utilisant la référence stockée. Ces méthodes ont un nom de la forme access$numero_unique et sont bien sûr inutilisables dans le code source puisqu'elles sont générées par le compilateur.

En tant que membre de la classe principale, une classe interne peut être déclarée avec le modificateur private ou protected.

Grâce au mot clé this, une classe peut faire référence dans le code source à son unique instance lors de l'exécution. Une classe interne possède au moins deux références :

l'instance de la classe interne elle−même

l'instance de sa classe principale

éventuellement les instances des classes internes imbriquées

Dans la classe interne, il est possible pour accéder à une de ces instances d'utiliser le mot clé this préfixé par le nom de la

classe suivi d'un point :

nom_classe_principale.this

nom_classe_interne.this

Le mot this seul désigne toujours l'instance de la classe courante dans son code source, donc this seul dans une classe interne désigne l'instance de cette classe interne.

Une classe interne non statique doit toujours être instanciée relativement à un objet implicite ou explicite du type de la classe principale. A la compilation, le compilateur ajoute dans la classe interne une référence vers la classe principale contenue dans une variable privée nommée this$0. Cette référence est initialisée avec un paramètre fourni au constructeur de la classe interne. Ce mécanisme permet de lier les deux instances.

La création d'une classe interne nécessite donc obligatoirement une instance de sa classe principale. Si cette instance n'est pas accessible, il faut en créer une et utiliser une notation

particulière de l'opérateur new pour pouvoir instancier la classe interne. Par défaut, lors de l'instanciation d'une classe interne, si aucune instance de la classe principale n'est utilisée, c'est l'instance courante qui est utilisée (mot clé this).

Exemple :

```
public class ClassePrincipale14 {
class ClasseInterne {
}
ClasseInterne ci = this. new ClasseInterne();
}
```

Pour créer une instance d'une classe interne dans une méthode statique de la classe principale, (la méthode main() par exemple), il faut obligatoirement instancier un objet de la classe principale avant et utiliser cet objet lors de la création de l'instance de la classe interne. Pour créer l'instance de la classe interne, il faut alors utiliser une syntaxe particulière de l'opérateur new.

Exemple :

```
public class ClassePrincipale15 {
class ClasseInterne {
}
ClasseInterne ci = this. new ClasseInterne();
static void maMethode() {
ClassePrincipale15 cp = new ClassePrincipale15();
ClasseInterne ci = cp. new ClasseInterne();
}
}
```

Il est possible d'utiliser une syntaxe condensée pour créer les deux instances en une seule et même ligne de code.

Exemple :

```
public class ClassePrincipale19 {
class ClasseInterne {
}
static void maMethode() {
ClasseInterne ci = new ClassePrincipale19(). new ClasseInterne();
}
}
```

Une classe peut hériter d'une classe interne. Dans ce cas, il faut obligatoirement fournir aux constructeurs de la classe une référence sur la classe principale de la classe mère et appeler explicitement dans le constructeur le constructeur de cette classe principale avec une notation particulière du mot clé super

Exemple :

```
public class ClassePrincipale9 {
public class ClasseInterne {
}
class ClasseFille extends
ClassePrincipale9.ClasseInterne {
ClasseFille(ClassePrincipale9 cp) {
cp. super();
}
}
}
```

Une classe interne peut être déclarée avec les modificateurs final et abstract. Avec le modificateur final, la classe interne ne pourra être utilisée comme classe mère. Avec le modificateur abstract, la classe interne devra être étendue pour pouvoir être instanciée.

2. Les classes internes locales

Ces classes internes locales (local inner−classes) sont définies à l'intérieur d'une méthode ou d'un bloc de code. Ces classes ne sont utilisables que dans le bloc de code où elles sont définies. Les classes internes locales ont toujours accès aux membres de la classe englobante.

Exemple :

```
public class ClassePrincipale21 {
int varInstance = 1;
public static void main(String args[]) {
ClassePrincipale21 cp = new
ClassePrincipale21(); cp.maMethode();
}
public void maMethode() {
class ClasseInterne {
public void affiche() {
System.out.println("varInstance = " + varInstance);
}
}
ClasseInterne ci = new ClasseInterne();
ci.affiche();
}
}
```

Résultat :

```
C:\testinterne>javac ClassePrincipale21.java
C:\testinterne>java ClassePrincipale21
varInstance = 1
```

Leur particularité, en plus d'avoir un accès aux membres de la classe principale, est d'avoir aussi un accès à certaines variables locales du bloc où est définie la classe interne.
Ces variables définies dans la méthode (variables ou paramètres de la méthode) sont celles qui le sont avec le mot clé final. Ces variables doivent être initialisées avant leur utilisation par la classe interne. Elles sont utilisables n'importe où dans le code de la classe interne.

Le modificateur final désigne une variable dont la valeur ne peut être changée une fois qu'elle a été initialisée.

Exemple :

```
public class ClassePrincipale12 {
public static void main(String args[]) {
ClassePrincipale12 cp = new
ClassePrincipale12(); cp.maMethode();
}
public void maMethode() {
int varLocale = 3;
class ClasseInterne {
public void affiche() {
System.out.println("varLocale = " + varLocale);
}
}
ClasseInterne ci = new ClasseInterne();
ci.affiche();
}
}
```

Résultat :

```
javac ClassePrincipale12.java
ClassePrincipale12.java:14: Attempt to use a non−final variable
varLocale from a different method. From enclosing blocks, only final local
variables are availab le.
System.out.println("varLocale = " + varLocale);
^
1 error
```

Cette restriction est imposée par la gestion du cycle de vie d'une variable locale. Une telle variable n'existe que durant l'exécution de cette méthode. Une variable finale est une variable dont la valeur ne peut être modifiée après son initialisation. Ainsi, il est possible sans risque

pour le compilateur d'ajouter un membre dans la classe interne et de copier le contenu de la variable finale dedans.

Exemple :

```
public class ClassePrincipale13 {
public static void main(String args[]) {
ClassePrincipale13 cp = new
ClassePrincipale13(); cp.maMethode();
}
public void maMethode() {
final int varLocale = 3;
class ClasseInterne {
public void affiche(final int varParam) {
System.out.println("varLocale = " + varLocale);
System.out.println("varParam = " + varParam);
}
}
ClasseInterne ci = new ClasseInterne();
ci.affiche(5);
}
}
```

Résultat :

```
C:\>javac ClassePrincipale13.java
C:\>java ClassePrincipale13
varLocale = 3
varParam = 5
```

Pour permettre à une classe interne locale d'accéder à une variable locale utilisée dans le bloc de code où est définie la classe interne, la variable doit être stockée dans un endroit accessible par la classe interne. Pour que cela fonctionne, le compilateur ajoute les variables nécessaires dans le constructeur de la classe interne.

Les variables accédées sont dupliquées dans la classe interne par le compilateur. Il ajoute pour chaque variable un membre privé dans la classe interne dont le nom est de la forme val$nom_variable. Comme la variable accédée est déclarée finale, cette copie peut être faite sans risque. La valeur de chacune de ces variables est fournie en paramètre du constructeur qui a été modifié par le compilateur.
Une classe qui est définie dans un bloc de code n'est pas un membre de la classe englobante : elle n'est donc pas accessible en dehors du bloc de code où elle est définie. Ses restrictions sont équivalentes à la déclaration d'une variable dans un bloc de code.
Les variables ajoutées par le compilateur sont préfixées par this$ et val$. Ces variables et le constructeur modifié par le compilateur ne sont pas utilisables dans le code source.
Etant visible uniquement dans le bloc de code qui la définit, une classe interne locale ne peut pas utiliser les modificateurs public, private, protected et static dans sa définition. Leur utilisation provoque une erreur à la compilation.

Exemple :

```
public class ClassePrincipale11 {
public void maMethode() {
public class ClasseInterne {
}
}
}
```

Résultat :

```
javac ClassePrincipale11.java
ClassePrincipale11.java:2: '}' expected.
public void maMethode() {
^
ClassePrincipale11.java:3: Statement expected.
public class ClasseInterne {
^
ClassePrincipale11.java:7: Class or interface declaration expected.
}
^
3 errors
```

3. Les classes internes anonymes

Les classes internes anonymes (anonymous inner−classes) sont des classes internes qui ne possèdent pas de nom. Elles ne peuvent donc être instanciées qu'à l'endroit où elles sont définies.

Ce type de classe est très pratique lorsqu'une classe doit être utilisée une seule fois : c'est par exemple le cas d'une classe qui doit être utilisée comme un callback.

Une syntaxe particulière de l'opérateur new permet de déclarer et instancier une classe interne :

```
new classe_ou_interface () {
  définition des attributs et des méthodes de la classe interne
}
```

Cette syntaxe particulière utilise le mot clé new suivi d'un nom de classe ou d'interface que la classe interne va respectivement étendre ou implémenter. La définition de la classe suit entre deux accolades. Une classe interne anonyme peut soit hériter d'une classe soit implémenter une interface mais elle ne peut pas explicitement faire les deux.

Si la classe interne étend une classe, il est possible de fournir des paramètres entre les parenthèses qui suivent le nom de la classe. Ces arguments éventuels fournis au moment de l'utilisation de l'opérateur new sont passés au constructeur de la super−classe. En effet, comme la classe ne possède pas de nom, elle ne possède pas non plus de constructeur.

Les classes internes anonymes qui implémentent une interface héritent obligatoirement de la classe Object. Comme cette classe ne possède qu'un constructeur sans paramètre, il n'est pas possible lors de l'instanciation de la classe interne de lui fournir des paramètres.

Une classe interne anonyme ne peut pas avoir de constructeur puisqu'elle ne possède pas de nom mais elle peut avoir des initialisateurs.

Exemple :

```
public void init() {
boutonQuitter.addActionListener(
new ActionListener() {
public void actionPerformed(ActionEvent e) {
System.exit(0);
```

```
}
}
);
}
```

Les classes anonymes sont un moyen pratique de déclarer un objet sans avoir à lui trouver un nom. La contrepartie est
que cette classe ne pourra être instanciée dans le code qu'à l'endroit où elle est définie : elle est déclarée et instanciée en un seul et unique endroit.
Le compilateur génère un fichier ayant pour nom la forme suivante : nom_classe_principale$numéro_unique. En fait, le compilateur attribut un numéro unique à chaque classe interne anonyme et c'est ce numéro qui est donné au nom du fichier préfixé par le nom de la classe englobante et d'un signe '$'.

4. Les classes internes statiques

Les classes internes statiques (static member inner−classes) sont des classes internes qui ne possèdent pas de référence vers leur classe principale. Elles ne peuvent donc pas accéder aux membres d'instance de leur classe englobante. Elles peuvent toutefois avoir accès aux variables statiques de la classe englobante.
Pour les déclarer, il suffit d'utiliser en plus le modificateur static dans la déclaration de la classe interne.
Leur utilisation est obligatoire si la classe est utilisée dans une méthode statique qui par définition peut être appelée sans avoir d'instance de la classe et que l'on ne peut pas avoir une instance de la classe englobante. Dans le cas contraire, le compilateur indiquera une erreur :

Exemple :

```
public class ClassePrincipale4 {
class ClasseInterne {
public void afficher() {
System.out.println("bonjour");
}
}
```

public static void main(String[] args) {
new ClasseInterne().afficher();
}
}
Résultat :
javac ClassePrincipale4.java
ClassePrincipale4.java:10: No enclosing instance of class ClassePrincipale4 is i n scope; an explicit one must be provided when creating inner class ClassePrinci pale4. ClasseInterne, as in "outer. new Inner()" or "outer. super()".
new ClasseInterne().afficher();
^
1 error

En déclarant la classe interne static, le code se compile et peut être exécuté.

Exemple :
public class ClassePrincipale4 {
static class ClasseInterne
{
public void afficher() {
System.out.println("bonjour");
}
}
public static void main(String[] args) {
new ClasseInterne().afficher();
}
}

Résultat :
javac ClassePrincipale4.java
java ClassePrincipale4
bonjour

Comme elle ne possède pas de référence sur sa classe englobante, une classe interne statique est traduite par le compilateur comme une classe principale. En fait, il est difficile de les mettre dans une catégorie (classe principale ou classe interne) car dans le code source c'est une classe interne (classe définie dans une autre) et dans le bytecode généré c'est une classe principale. Ce type de classe n'est pas très employé.

5. Les membres static dans une classe interne

Historiquement, une erreur est émise par le compilateur si une classe interne déclare un membre static qui n'est pas une constante.

Exemple (code Java 15) :

```
public class TestStatic {
public void traiter() {
class MaClasse {
public final static int valeur = 0;
public static void afficher() {
System.out.println(valeur);
}
};
}
}
```

Résultat :

```
C:\java>javac TestStatic.java
TestStatic.java:7: error: Illegal static declaration in inner class
MaClasse public static void afficher() {
^
modifier 'static' is only allowed in constant variable declarations
1 error
```

Cela implique qu'une classe interne ne peut déclarer un membre soit la définition record puisque les records imbriqués sont implicitement static.

Exemple (code Java 15) :

```
public class TestInnerRecord {
class MaClasse {
record MonRecord(String nom) {};
};
}
```

Résultat :

```
C:\java>javac −version
javac 15
C:\java>javac TestInnerRecord.java
TestInnerRecord.java:4: warning: 'record' may become a restricted type name in a future
release and may be unusable for type declarations or as the element type of an array
record MonRecord(String nom) {};
^
TestInnerRecord.java:4: error: cannot find symbol
record MonRecord(String nom) {};
^
symbol:       class record
location: class TestInnerRecord.MaClasse
1 error
1 warning
```

La JEP 395 introduite dans Java 16, permet à une classe interne de déclarer des membres qui soient implicitement ou explicitement static.

Exemple (code Java 16) :

```
public class TestStatic {
public void traiter() {
class MaClasse {
```

```
public static int valeur = 0;
public static void afficher() {
System.out.println(valeur);
}
};
}
}
```

Résultat :

```
C:\java>javac TestStatic.java
C:\java>
```

Cela permet notamment à une classe interne de définir des membres qui soient la définition d'un record.

Exemple (code Java 16) :

```
public class TestInnerRecord {
class MaClasse {
record MonRecord(String nom) {};
};
}
```

Résultat :

```
C:\java>javac −version
javac 16.0.1
C:\java>javac TestInnerRecord.java
C:\java>
```

IX. La gestion dynamique des objets

Tout objet appartient à une classe et Java sait la reconnaitre dynamiquement.

Java fournit dans son API un ensemble de classes qui permettent d'agir dynamiquement sur des classes. Cette technique est appelée introspection et permet :

de décrire une classe ou une interface : obtenir son nom, sa classe mère, la liste de ses méthodes, de ses variables de classe, de ses constructeurs et de ses variables d'instances

d'agir sur une classe en envoyant à un objet Class des messages comme à tout autre objet. Par exemple, créer dynamiquement à partir d'un objet Class une nouvelle instance de la classe représentée