

JAVA OBJET

Michel MASSON

Ce document décrit les principales caractéristiques objet du langage Java ; on supposera connus les éléments de base du langage (cf. [1]). Il peut être complété par [3] et en particulier « Thinking in java - B. Eckel» ou sa traduction française qui constituent des documents de référence.

- [1] Informatique Vol. 1,2 &3 F. Rossi <http://docs.ufrmd.dauphine.fr/java/polys/>
- [2] Thinking in Java. B. Eckel <http://penserenjava.free.fr/>
- [3] Java Autoformation I. Valembois & L. Millecam - Ed. Ellipses
- [4] Java: la synthèse G. Clavel, N. Mirouze, ... Ed. Dunod
- [5] Java : La Maîtrise (Java 5 & 6) J. Bougeault - Ed. Eyrolles
- [6] Java : Exercices en Java Cl. Delannoy - Ed. Eyrolles

Conventions :

Les exemples s'appuient sur l'environnement de développement Eclipse téléchargeable sur www.eclipse.org

Chaque mot correspondant à un concept nouvellement introduit est typographié en **gras**.

Les mots-clés du langage et les programmes utilisent cette police.

1. Introduction.

Java a été conçu en 1994 par la société SUN. Son succès s'explique par plusieurs raisons: c'est la première plateforme gratuite, maintenue, fiable et sécurisée; à cela s'ajoute la portabilité des applications. Un autre point décisif fut l'introduction des **servlets** ouvrant la voie à la programmation au sein des serveurs et constituant une amélioration par rapport à la programmation CGI (Common Gateway Interface). Les distributions successives (Java Development Kit) ont apporté de nombreuses améliorations, Java 2 désigne les versions 1.2 et supérieures (version actuelle Juin 2014: 1.8.x)

2 . La notion d'Objet.

La programmation dans quelque langage que ce soit procède par abstractions. Ainsi tout langage de programmation est une abstraction de la machine sur laquelle il est installé, et tout problème doit être modélisé (résultat de l'abstraction) pour être représenté dans un langage de programmation. Cette dernière étape est la plus délicate car la modélisation peut amener le programmeur à s'écartez considérablement du problème initial.

La méthodologie objet procède différemment: le problème initial est modélisé à l'aide d'objets. Or l'objet est le constituant de base d'un langage à objets, le programmeur fait alors correspondre à chaque objet du modèle un objet du langage. Bien sûr, les objets du programme sont plus nombreux car il faut prendre en compte les objets liés aux tâches système, mais le principe reste simple: tout problème correspond à un ensemble d'objets que le programmeur transpose en objets du langage. Toute modification du problème initial correspond à un ajout et/ou à une modification des objets du programme.

Le principe de base de la programmation objet tient en une phrase: "Tout est objet" (on parle de réification). Tout problème peut se décomposer en éléments eux-mêmes représentables en objets; un objet est caractérisé par un ensemble d'informations qui lui sont propres. Ces informations se subdivisent en deux: les **attributs** qui décrivent les caractéristiques de l'objet et les **méthodes** qui décrivent les traitements qui peuvent être appliqués à l'objet.

Parmi les objets, on distingue les classes et les instances: les classes sont des objets génériques décrivant un ensemble d'objets semblables, les instances. Tout objet est donc obtenu par instantiation de sa classe grâce à la méthode new. Les classes elles-mêmes sont des instances de la classe `Class` mais sont gérées par la machine virtuelle Java.

Tout objet possède un type (sa classe), les classes peuvent être hiérarchisées grâce à la relation **d'héritage**. Dans tout ce qui suit, les liens d'héritage seront représentés par une flèche simple et les liens d'instanciation par une flèche double.

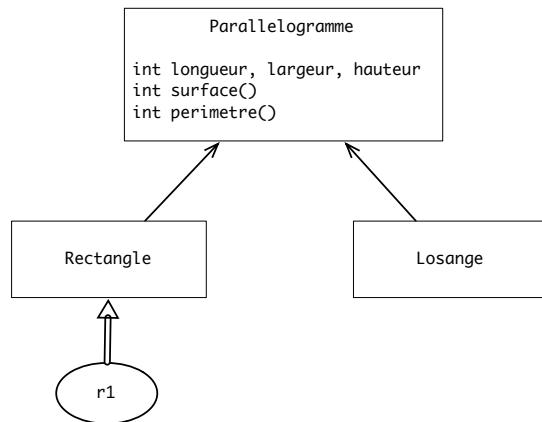
L'exemple ci-contre décrit l'espace des figures géométriques, la classe *parallelogramme* comprend trois attributs: *longueur*, *largeur* et *hauteur* ainsi que deux méthodes *périmètre* et *surface*: cette classe est dérivée en deux sous-classes: *Losange* et *Rectangle*. Une instance *r1* de la classe *Rectangle* est ensuite ajoutée à la hiérarchie.

Le monde réel offre une infinité d'exemples: une voiture est un objet dont les attributs sont *numéro_serie*, *propriétaire*, *marque*, *modèle*, *puissance*,... . Mais un évènement temporel peut également être considéré comme un objet: un accident automobile peut être décrit par sa nature, sa date, les véhicules concernés; si l'accident est bénin, chaque protagoniste pourra repartir et l'existence de l'objet correspondant sera virtuelle.

Les liens dynamiques entre objets sont assurés par l'envoi de message: les objets communiquent entre eux par envoi de message (l'objet o_i répond au message m_j envoyé par un autre objet en activant une méthode de sa classe de nom m_j : si la classe de o_i ne possède pas cette méthode, la hiérarchie d'héritage est interrogée).

Tout langage objet repose sur 4 principes:

- La réification ("tout est objet")
- La notion de classe (tout objet a un type, i.e. tout objet est instance d'une classe)
- L'héritage (les classes sont organisées hiérarchiquement)
- Tout objet communique avec les autres objets par l'envoi de message



3 . Les classes dans le langage Java.

Le concept de classe est fondamental en programmation objet, il permet de rassembler (factoriser) les propriétés communes à un ensemble d'objets sous forme de variables et de méthodes.

L'exemple 3.1 fait intervenir 3 classes: Ville, Capitale et Prefecture, une instance de la classe Ville est caractérisée par les attributs nom et population (en millions d'habitants); c'est la classe de **base**. La classe Ville est dérivée en 2 sous-classes Capitale et Prefecture, elles possèdent chacune un attribut supplémentaire: pays et département. Chaque classe possède une méthode `affiche()` dont le résultat est l'impression de ces caractéristiques sur le flux de sortie standard.

La classe Ex31 contient la méthode `main()`; c'est elle qui est lancée au démarrage (cf. § 3.2), elle crée une instance de chaque classe (v, c et p) et envoie la méthode `affiche()` à ces 3 instances; le code de ces 4 classes correspond à:

Exemple 3.1

```
package exemplesPoly.ex31;
class Ville{
    String nom;
    float population;
    void affiche(){
        System.out.println("Ville: " + nom +
                           " nombre d'habitants: " + population);
    }
}
class Capitale extends Ville{
    String pays;
    void affiche(){
        super.affiche();
        System.out.println("capitale de: " + pays );
    }
}
class Prefecture extends Ville{
    String departement;
    void affiche(){
        super.affiche();
        System.out.println("Préfecture de: "+departement );
    }
}
public class Ex31{
    public static void main(String[] args){
        Ville v = new Ville();
        v.nom = "Lyon";
        v.population = 0.46f;
        Capitale c = new Capitale();
        c.nom = "Paris";
        c.population = 2.2f;
        c.pays = "France";
        Prefecture p = new Prefecture();
        p.nom = "Caen";
        p.population = 0.22f;
        p.departement = "Calvados";
        v.affiche();c.affiche();p.affiche();
    }
}
```

```
Résultats de l'exécution de ce code:
Ville: Lyon nombre d'habitants: 0.46
Ville: Paris nombre d'habitants: 2.2
capitale de: France
Ville: Caen nombre d'habitants: 0.22
Préfecture de: Calvados
```

Remarque: le mot-clé **super** est important, partout où il est rencontré il doit être remplacé par la classe de base (i.e. la classe parent); ainsi dans la méthode **affiche()** de la classe **Capitale**, **super.affiche()** désigne la méthode **affiche()** de la classe **Ville** à laquelle on ajoute ici une instruction pour assurer l'impression du pays.

3.1 Classes et constructeurs.

Le **constructeur** est un moyen efficace pour construire les instances d'une classe; c'est une méthode qui a le nom de sa classe, son appel crée une instance de la classe. Examinons les modifications apportées à l'exemple 3.1:

Exemple 3.1.1

```
package exemplesPoly.ex311;
class Ville{
    String nom;
    float population;
    Ville(String nom, float population){
        this.nom = nom;
        this.population = population;
    }
    void affiche(){
        System.out.println("Ville: " + nom +
                           "nombre d'habitants: " + population);
    }
}
class Capitale extends Ville{
    String pays;
    Capitale(String n, float p, String pays){
        super(n,p);this.pays = pays;
    }
    void affiche(){
        super.affiche();
        System.out.println("capitale de: " + pays );
    }
}
class Prefecture extends Ville{
    String departement;
    Prefecture(String n, float p, String d){
        super(n,p);departement = d;
    }
    void affiche(){
        super.affiche();
        System.out.println("Préfecture de: " +departement);
    }
}
public class Ex311{
    public static void main(String ){
        Ville v = new Ville("Lyon",0.46f);
        Capitale c = new Capitale("Paris",2.2f,"France");
        Prefecture p = new Prefecture("Caen",0.22f,"Calvados");
        v.affiche();c.affiche(); p.affiche();
    }
}
```

Nous avons ajouté aux 3 premières classes un constructeur, il permet d'initialiser l'instance en utilisant les arguments passés en paramètres, de plus il est possible d'utiliser le constructeur de

la classe de base (i.e. la classe-mère) grâce au mot-clé **super**, cela évite de répéter les instructions de cette classe. Dans ce cas, il doit **impérativement** être en tête. En fait, toute classe possède au moins un constructeur; lorsque celui-ci n'est pas explicité un **constructeur par défaut** est ajouté par l'interpréteur: il alloue un espace mémoire pour l'instance et initialise par défaut les attributs de l'instance (0 pour les attributs numériques, "" pour les chaînes de caractères, **null** pour les références). Ainsi en Java, tout constructeur fait **obligatoirement** appel de façon explicite ou implicite au constructeur de la classe mère. Une classe peut avoir plusieurs constructeurs, pourvu que leurs signatures soient différentes (la **signature** d'une méthode est constituée de son nom suivi de la liste du type de ses arguments). Attention! L'ajout d'un ou plusieurs constructeurs par le programmeur supprime le constructeur par défaut, il sera parfois nécessaire de le rajouter (cf. la classe A de l'ex. 3.1.2).

Ordre d'appel des constructeurs et construction des instances:

La création d'une instance peut faire appel à plusieurs constructeurs si la classe instanciée hérite de plusieurs classes; le constructeur le plus général est d'abord appelé, l'instanciation se termine avec l'appel du constructeur local. Ainsi tout appel d'un constructeur correspond aux étapes suivantes:

1. réservation de mémoire
2. appel récursif des constructeurs (de la classe locale, de la classe **super**, . . .)
3. initialisation des attributs
4. exécution des constructeurs (en commençant par le dernier appelé)

Ces étapes sont illustrées avec l'exemple suivant:

Exemple 3.1.2

```
package exemplesPoly.ex312;
public class Ex312 {
    public static void main(String[] args) {
        Test t= new Test();
        t.truc();
    }
}
class A{
    A(int i){System.out.println("Constr. A " + i);}
    A(){System.out.println("Constr. A par déf.");}
}
class B extends A{
    B(int i){
        super(i); // si supprimé, le constr. par déf.
        // de A est appelé
        System.out.println("Constr. B " + i);
    }
}
class C extends B{
    C(int i){
        super(i);
        System.out.println("Constr. C " + i);
    }
}

class Test{
    C c1=new C(1);
    Test(){System.out.println("Constr. Test");}
    void truc(){ System.out.println("fini !!!");}
    C c2=new C(2);
    C c3=new C(3);
}
```

```
résultats
Constr. A 1
Constr. B 1
Constr. C 1
Constr. A 2
Constr. B 2
Constr. C 2
Constr. A 3
Constr. B 3
Constr. C 3
Constr. Test
fini !!!
```

Remarque:

le programme principal construit une instance `t` la classe `Test`, les initialisations de `c1`, `c2` et `c3` sont effectuées en premier, elles sont suivies de l'appel du constructeur local (il n'y a pas d'appel récursif des constructeurs pour `Test`) puis de l'appel de la méthode `truc()`.

Il se dégage de l'exemple précédent que la construction d'une instance est incrémentale: une instance de la classe C est obtenue en construisant d'abord une instance de la classe A à laquelle on ajoute les spécifications de la classe B puis on ajoute les spécifications de la classe C. L'exemple 4.1 met à jour ce mécanisme en dissociant l'héritage des méthodes de celui des attributs.

3.2 Liens classe – fichiers .

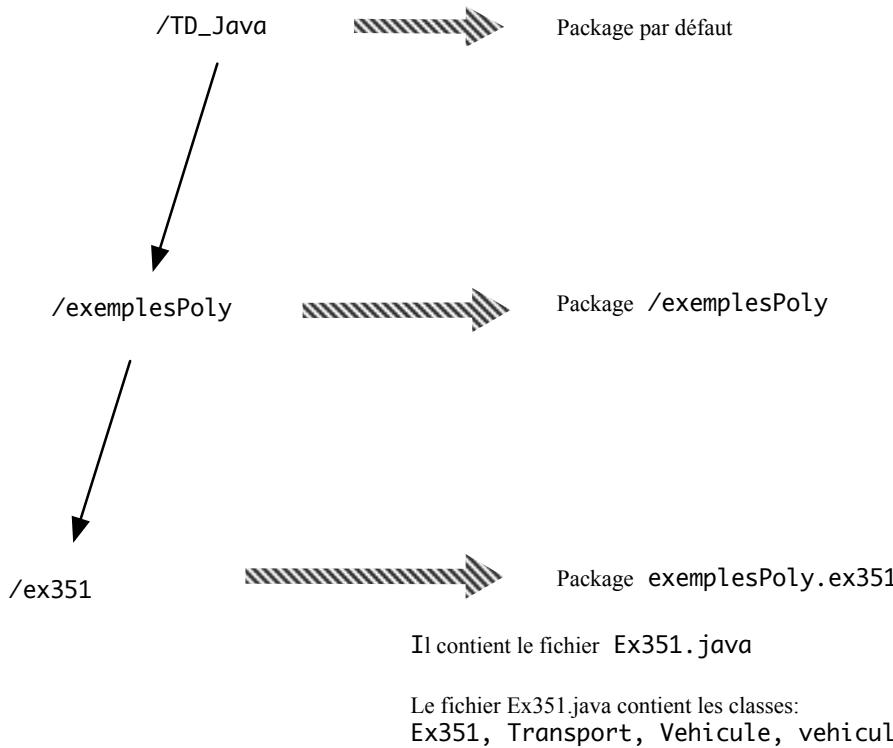
Une application Java est représentée par une ou plusieurs classes. Toute classe Java est en principe associée à un fichier dont le nom est celui de la classe avec l'extension `.java`, dans ce cas la classe est déclarée `public` (cf. § suivant). Mais on peut rassembler plusieurs classes dans un seul fichier à la condition qu'une seule classe soit déclarée `public`. Une des classes doit contenir la méthode `main` ; c'est elle qui est lancée au démarrage de l'application. Rappelons quelques règles de la programmation Java :

- `//` permet d'insérer une ligne de commentaires (en fait tous les caractères qui suivent sont assimilés à des commentaires jusqu'au retour-chariot). Un passage situé entre les séquences `/*` et `*/` est considéré comme un commentaire.
- Les noms de classes commencent par une majuscule, il est d'usage dans les mots composés de marquer la transition entre les mots par une majuscule. `numero`, `numeroVehicule` désignent ainsi des noms de variables d'instances ou de méthode ; `Vehicule`, `VehiculeTest` désignent des noms de classe.

3.3 Les Packages (espaces de nom): un outil pour gérer les classes.

À la différence de C++ où les classes sont traitées uniformément, une classe Java peut être définie spécifiquement pour un utilisateur grâce aux espaces de noms. Un espace de noms repose sur une hiérarchie de répertoires calquée sur le système d'exploitation hôte. Comme les répertoires du système hôte, les espaces de nom sont hiérarchisés (un espace de nom est une suite d'espaces de nom séparés par des points):

exemplesPoly.ex351
correspond au répertoire :
TD_Java/exemplesPoly/ex351
où TD_java est le répertoire projet regroupant plusieurs applications



Un espace de noms est donc un répertoire dans une arborescence de répertoires du système hôte, la racine de cette arborescence (/) correspond au package par défaut. Toute application écrite en java correspond à un ensemble de classes que l'utilisateur place dans un package grâce au mot-clé `package` suivi du nom du package, cette instruction doit être placée en tout début de programme. En son absence, l'ensemble des classes est placé dans le package par défaut. Sur la plate-forme Eclipse, les espaces de noms sont regroupés en projet. La racine du projet correspond au répertoire par défaut. L'utilisation de l'espace de nom permet de désigner sans ambiguïté une classe:

- à l'intérieur de son espace de nom, toute classe est référençable par son nom (i.e. son nom relatif)
- à l'extérieur de son espace de nom, la classe doit être référencée par son nom absolu:
`exemplesPoly.ex351.Vehicule` désigne la classe `Vehicule` située dans le package
`exemplesPoly.ex351`

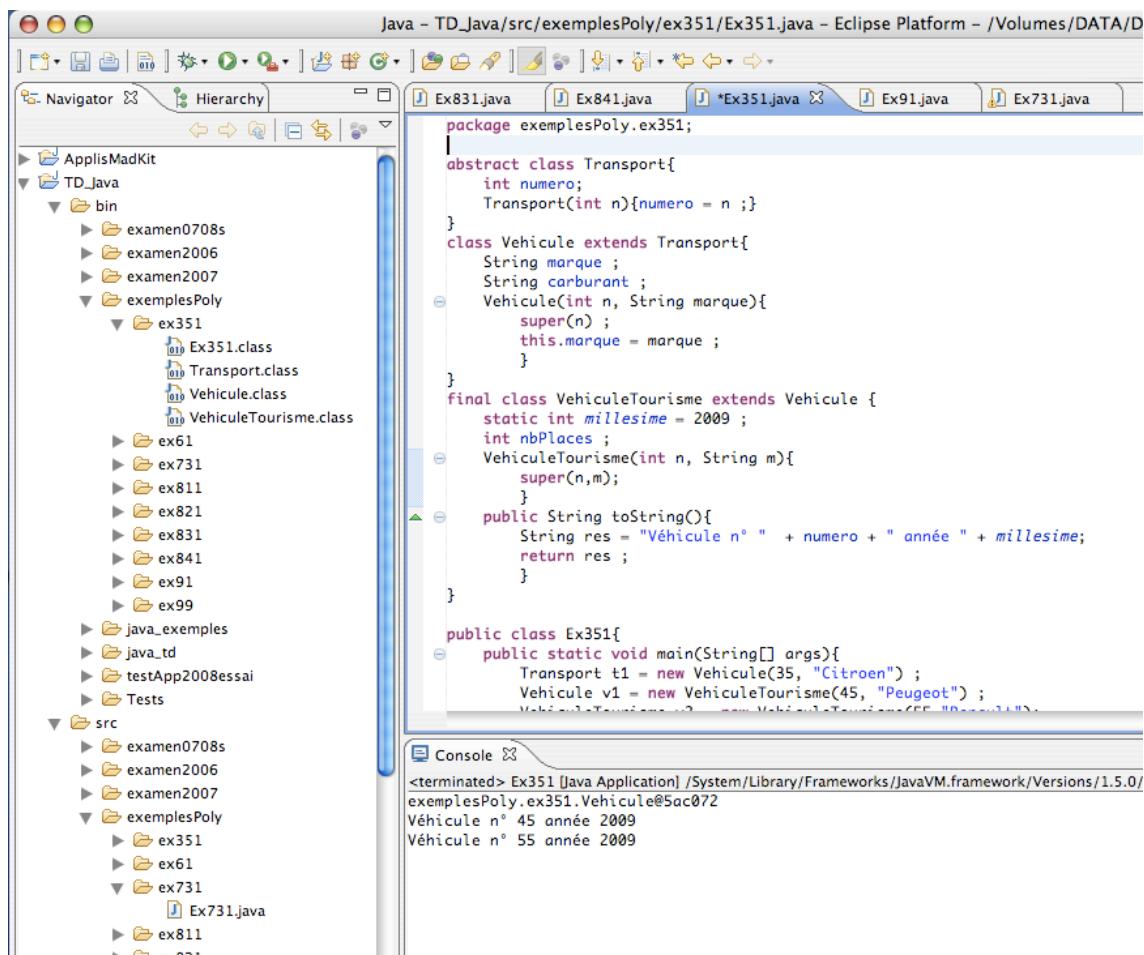
la construction d'un package peut nécessiter la présence de classes figurant dans un ou plusieurs autres packages. Plutôt que de répéter la définition de ces classes, il suffit de les importer à l'aide de l'instruction `import` (toujours placé en début de programme):

<code>import exemplesPoly.ex731;</code>	importe toutes les classes du package <code>exemplesPoly.ex731</code>
<code>import exemplesPoly.ex731.Proprietaire;</code>	importe uniquement la classe <code>Proprietaire</code> du package <code>exemples.ex731</code>

Le langage Java fournit des classes prédéfinies elles-mêmes réparties dans des packages, ils devront être importés en cas de besoin:

java.lang	Package importé par défaut: il contient les classes de base du langage: Object, String,...
java.util	Ce package contient les classes manipulant les collections, les classes Date, Time,...
java.applet	Classes utilisées pour la construction d'applets.
java.awt	Classes utilisées pour la réalisation d'interfaces graphiques.
java.awt.event	Classes utilisées pour la gestion d'événements: ActionEvent, FocusEvent,....

L'image suivante montre une recopie d'écran de la plate-forme Eclipse. A gauche une fenêtre d'affichage des packages: deux projets sont affichés (ApplisMadKit et TD_Java), les fichiers sources (.java) et les fichiers compilés (.class) figurent dans des répertoires séparés, le fichier Ex351.java se trouve dans le package exemplespoly.ex351 du projet TD_Java. La fenêtre de droite affiche une vue du fichier sélectionné, la fenêtre du bas affiche les résultats de l'exécution.



3.4 Spécificateurs d'accès. Facteur de visibilité d'une classe :

Les mots-clés `public`, `package` et `protected` permettent de gérer la **visibilité** d'une classe vis-à-vis des autres classes. Une classe déclarée `public` est accessible depuis tous les espaces de nom, une classe sans facteur de visibilité (ou avec le mot-clé `package` – facteur de visibilité par défaut) est visible des seules classes du package où elle figure. Une classe déclarée `protected` est visible de toutes les classes qui en dérivent (i.e. les classes filles).

Important: il n'est pas possible de restreindre la visibilité d'un membre au travers de l'héritage. Ainsi une méthode déclarée `public` ne peut pas être redéfinie `private` dans une classe dérivée.

Une classe déclarée `abstract` ne peut pas être instanciée, elle peut par contre posséder un constructeur (il sera appelé lors de l'appel récursif des constructeurs parent lors d'une instanciation). Une classe déclarée `final` ne peut pas être dérivée (en sous-classes), elle peut seulement être instanciée. Ces concepts sont repris dans l'exemple 3.5.1.

3.5 Attributs et Méthodes :

Un attribut est caractérisé par sa visibilité (cf. § précédent), sa **nature**, son **type**, son nom et une affectation (facultative). Une méthode est caractérisée de la même façon à l'exception de l'affectation, il faut préciser en outre la liste de ses arguments (leur portée est limitée au corps de la méthode) et le corps de la méthode. Rappelons que la **signature** d'une méthode est définie par son nom suivi de la liste du type de ses arguments.

La **nature** d'un attribut ou d'une méthode est indiquée par le mot-clé `static` ; sa présence indique que la méthode ou l'attribut sont des méthodes de classes ou des attributs de classe. L'absence de ce mot-clé indique qu'on a affaire à une variable (i.e. attribut) d'instance ou à une méthode d'instance. La valeur d'un attribut `static` est la même pour toutes les instances. Une méthode de classe est une méthode qui ne peut être envoyée qu'à une classe.

Le type d'un attribut renseigne sur les valeurs que peut prendre celui-ci ; il peut donc valoir un type primitif (cf. [1]), une classe ou une interface (cf. § 5).

Une méthode déclarée `abstract` ne possède pas de corps, il devra être spécifié dans la ou les classes dérivées. Une variable déclarée `final` ne peut pas être modifiée (lorsqu'elle est initialisée, sa valeur ne peut plus changer), on peut utiliser cette caractéristique pour déclarer une constante. Une méthode déclarée `final` ne peut pas être redéfinie.

Classe abstraite, elle ne peut être instanciée, mais elle peut posséder un constructeur.

Ex. 3.5.1

```
package exemplesPoly.ex351;

abstract class Transport{
    int numero;
    static int numeroCourant = 50;
    Transport(){numero = ++numeroCourant;}
}
class Vehicule extends Transport{
    String marque ;
    String carburant ;
    Vehicule(String marque){
        super() ;
        this.marque = marque ;
    }
}
final class VehiculeTourisme extends Vehicule {
    static int millesime = 2013 ;
    int nbPlaces ;
```

Les 4 classes de l'exemple sont incorporées dans le package : `exemplesPoly.ex351`

Le constructeur fait appel au constr. de la classe mère, cet appel DOIT être placé en tête.

Classe final: elle ne peut être dérivée.

L'utilisation de `this` permet de lever la confusion entre l'argument `marque` et l'attribut `marque`

```

VehiculeTourisme(String m){
    super(m);
}

public String toString(){
    String res="Véhicule "+marque+" n° " + numero;
    return res ;
}

public class Ex351{
    public static void main(String[] args){
        Transport t1 = new Vehicule("Renault") ;
        Vehicule v1 = new VehiculeTourisme("Peugeot");
        System.out.println(t1) ;
        System.out.println(v1) ;
    }
}

```

Résultat de l'exécution:
exemplesPoly.ex351.Vehicule@57f0d
Véhicule Peugeot n°52

Bien qu'abstraite, la classe Transport peut servir de type

Cette méthode est une redéfinition de la méthode publique `toString()` de la classe `Object`; il ne faut pas restreindre sa visibilité: elle doit être `public`.

La méthode `toString()` de `Vehicule` n'a pas été redéfinie, c'est la forme héritée de `Object` qui s'exécute

Commentaires :

La visibilité des trois premières classes n'est pas précisée, c'est donc celle par défaut qui prévaut: ces classes sont visibles des autres classes du package où elles sont incorporées (package `exemplesPoly.ex351`).

Le mot-clé `extends` indique le lien de hiérarchie entre les classes (ici la classe `Vehicule` hérite de la classe `Transport`).

La variable (attribut) `millesime` est une variable de classe, sa valeur initialisée à 2013 sera la même pour toutes les instances de la classe `VehiculeTourisme` (remarquons que l'initialisation n'est pas obligatoire).

Le constructeur de la classe `Vehicule` utilise le constructeur de sa classe-mère ; une telle écriture n'est pas obligatoire et aurait pu être remplacée par la définition suivante (plus lourde) :

```

Vehicule(String marque){
    numero = ++numeroCourant ;
    this.marque = marque ;
}

```

Ces 4 classes sont incorporées dans un même fichier, une des classes doit être déclarée `public` (classe `Test`).

4 . Héritage et polymorphisme

Les classes sont organisées hiérarchiquement, cela permet la factorisation des connaissances: un membre (attribut ou méthode) commun à deux classes C1 et C2 doit être placé dans une classe C dont C1 et C2 hériteront.

Notation : C sera alors appelée classe de base et C1 et C2 classes dérivées.

L'héritage concerne les attributs et les méthodes; l'héritage de méthode est dynamique (c'est à l'exécution du programme que la machine virtuelle Java décide de la méthode à appliquer) tandis que l'héritage d'attribut est statique, les instances gardent la trace de la façon dont elles ont été construites. L'exemple suivant illustre ces principes:

Ex. 4.1

```
package exemplesPoly.ex41;

class Base{
    public int n;
    public Base(int x){
        n = x;
    }
    public String toString(){
        return "Base " + n;
    }
    void test(){
        System.out.println("Base test " + this.n);
    }
}
class Complement extends Base{
    public int n;
    public Complement(int x, int y){
        super(x);
        this.n = y;
    }
    public String toString(){
        return "Complément " +
    }
    void test(){
        super.test();
        System.out.println(
            "Complement test hér. attr. "+this.n + " " + super.n +
            " test hér. méth. " + this.toString() + " " +
            super.toString());
    }
}
public class Ex41{
    public static void main(String[] args) {
        Complement c = new Complement(1,3);
        c.test();
    }
}

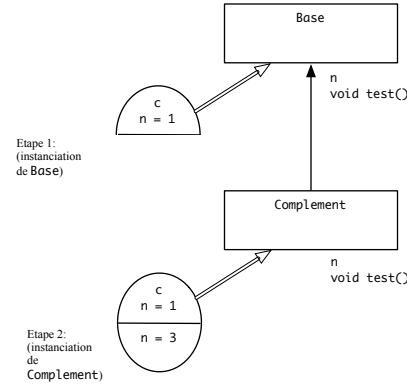
Résultats de l'exécution:
Base test 1
Complement test hér. attr. 3 1 test hér. méth. Complément 3 Base 1
```

C'est la valeur de n dans C (considérée comme une instance de la classe de base Base) qui est utilisée

C'est la valeur de n dans la classe dérivée qui est utilisée

Commentaires:

l'instance `c` de la classe `Complement` est construite incrémentalement. Le constructeur le plus "haut" est d'abord invoqué, puis les autres constructeurs viennent compléter cette construction. Dans cet exemple, la variable `n` est définie dans la classe de base (`Base`) et dans la classe dérivée (`Complement`), sa valeur change selon que `c` est considérée comme une instance de `Base` ou comme une instance de `Complement`. La figure ci-contre montre la construction de l'instance `c`: elle est d'abord construite comme une instance de `Base`, puis elle est complétée pour former une instance de `Complement`.



Deux mécanismes importants sont liés à l'héritage de méthode: la redéfinition et la surcharge.

- Une méthode `m` d'une classe dérivée **redéfinit** la méthode `m` de la classe de base si elles ont la même signature (seul le corps est modifié).
- Une méthode `m` d'une classe dérivée **surcharge** la méthode `m` de la classe de base si elles ont des signatures différentes.

Attention ! Ces deux mécanismes apparemment proches sont fondamentalement différents. La surcharge est gérée à la compilation par la machine virtuelle Java tandis que la redéfinition est gérée dynamiquement à l'exécution : elle participe à la mise en œuvre du polymorphisme de méthode (cf. chapitre suivant).

4.1 Transtypage:

Rappelons que tout objet peut être typé par son propre type ou par un autre type. On parlera de transtypage ascendant (upcasting) lorsque l'objet est typé avec le type d'une classe supérieure et de transtypage descendant (downcasting) dans le cas inverse. Cette opération de transtypage (aussi appelée cast) est réalisée en faisant précéder l'objet de son nouveau type placé entre parenthèses. Examinons les résultats obtenus en modifiant la classe `Ex351` de l'exemple 3.5.1:

Ex. 4.1.1

```

public class Ex411{
    public static void main(String[] args){
        Transport t1 = new Vehicule(35, "Citroen");
        Transport t2 = new VehiculeTourisme(45, "Peugeot");
        VehiculeTourisme v1 = (VehiculeTourisme) t2;
        //VehiculeTourisme v2 = t2;

        System.out.println(t1);
        System.out.println(t2);
    }
}

Résultat de l'exécution:
exemplesPolys.ex411.Véhicule@5ac072
Véhicule n°45 année 2009

```

Transtypage ascendant

Transtypage descendant, le cast (transtypage) est obligatoire.

L'absence de transtypage de cette instruction provoque une erreur de compilation.

Commentaires :

La première instanciation est un upcasting de l'objet `t1` vers un type supérieur, l'opérateur de cast n'est pas obligatoire (une instance de la classe `Vehicule` est d'abord une instance de la classe `Transport` - cf. l'ordre d'appel des constructeurs - ce qui dispense de l'opérateur de cast).

La troisième instanciation est un downcasting de l'objet `t2` de type `Transport` vers un type inférieur: l'opérateur de cast est alors obligatoire.

4.2 Le polymorphisme ou affectation tardive (late binding)

Le polymorphisme est un concept puissant offert par la programmation objet qui permet de séparer le "quoi" du "comment". Il va permettre d'effectuer des opérations sur des objets de type différents comme s'ils étaient du même type.

Le problème fondamental de la programmation objet est de trouver la bonne méthode à appliquer lors de l'exécution d'une instruction `o.m()` où `m()` est une méthode appliquée à un objet `o` de type `T`. On observera dans l'exemple suivant que chaque classe du package correspond à un fichier (elles sont toutes public).

Ex. 4.2.1

```
package exemplesPoly.ex421;
public class Figure{
    void affiche(){
        System.out.println("Affichage d'une figure") ;
    }
}

package exemplesPoly.ex421;
public class Rectangle extends Figure{
    void affiche(){
        System.out.println("Affichage d'un rectangle") ;
    }
}

package exemplesPoly.ex421;
public class Cercle extends Figure{
    void affiche(){
        System.out.println("Affichage d'un cercle") ;
    }
}

package exemplesPoly.ex421;
public class Ex421{
    public static void main(String[] args){
        Figure[] listeFigures= {new Rectangle(),
                               new Cercle(),
                               new Rectangle() };
        afficheListe(listeFigures) ;
    }
    static void afficheListe(Figure[] listeFigures){
        for(int i=0 ;i<3 ;i++){
            if(listeFigures [i] instanceof Rectangle){
                Rectangle x = (Rectangle) listeFigures[i] ;
                x.affiche() ;
            }
            if(listeFigures [i] instanceof Cercle){
                Cercle x = (Cercle) listeFigures[i] ;
```

```
        x.affiche() ;  
    }  
}
```

Résultats:
Affichage d'un rectangle
Affichage d'un cercle
Affichage d'un rectangle

Commentaires :

listeFigures[] qui est de type Figure contient des objets de type Rectangle ou Cercle, la méthode afficheListe() a pour fonction d'afficher les objets du tableau passé en arguments en fonction de leur type. Pour cela, la méthode doit vérifier le type avant de les downcaster pour l'appel de la bonne méthode d'affichage de la figure.

Définissons un nouveau package `Ex421polym` où la classe `Ex421` est réécrite en `Ex421polym` après importation des classes `Figure`, `Rectangle` et `Cercle` du package précédent:

```
package exemplesPoly.ex421polym;
import exemplesPoly.ex421.Figure;
import exemplesPoly.ex421.Rectangle;
import exemplesPoly.ex421.Cercle;

public class Ex421polym{
    public static void main(String[] args){
        Figure[] listeFigures = {new Rectangle(),
                                  new Cercle(),
                                  new Rectangle()};
        afficheListe(listeFigures) ;
    }
    static void afficheListe(Figure [] listeFigures){
        for(int i=0 ;i<listeFigures.length;i++){
            listeFigures[i].affiche();
        }
    }
}
```

```
Résultats:  
Affichage d'un rectangle  
Affichage d'un cercle  
Affichage d'un rectangle
```

Ces instances sont rangées dans un tableau de type Figure.

La méthode `affiche()` est envoyée à un tableau de type `Figure`.

La "bonne" méthode `affiche()` est reçue par chaque instance.

Commentaires :

On obtient les mêmes résultats ! L'explication tient au choix de la méthode à appliquer qui est fait à l'exécution (late binding) ; une méthode `affiche()` est envoyée à des objets qui ont tous le type `Figure`, à l'exécution la machine virtuelle Java a accès aux objets et constate que ces objets sont des cercles ou des rectangles et qu'une méthode redéfinie existe pour chacun de ces objets.

L'intérêt du polymorphisme est double : la méthode `afficheListe()` est devenue indépendante des objets à afficher et on peut ajouter de nouvelles catégories d'objets (par ex. des parallélogrammes) sans avoir à la modifier !

5 . Les Interfaces :

Avec les interfaces Java propose un mécanisme efficace de gestion des classes. La forme générale d'une interface est :

```
Interface exempleInterface{
    type_meth1 meth1(arguments);
    type_meth2 meth2(arguments);
}
```

Une interface s'apparente à une classe abstraite dont les méthodes seraient abstraites, elle définit un schéma à respecter pour toute classe implémentant cette interface. Le mot-clé `implements` inséré dans la définition d'une classe oblige cette dernière à posséder toutes les méthodes définies dans l'interface. Une classe peut implémenter plusieurs interfaces. A la différence d'une classe, une interface ne possède pas de constructeur, elle ne peut donc pas être instanciée. Une interface peut servir à typer des variables, des paramètres de méthode ou le résultat d'une méthode.

L'exemple précédent peut être réécrit avec une interface: toutes les figures doivent posséder une méthode `affiche()`, on peut donc définir une interface incluant la déclaration de cette méthode, on obtient le programme suivant :

Ex. 5.1

```
package exemplesPoly.ex51 ;

class Figure implements Affichable{
    public void affiche(){
        System.out.println("Affichage d'une figure");
    }
}
class Rectangle extends Figure{
    public void affiche(){
        System.out.println("Affichage d'un rectangle");
    }
}
class Cercle extends Figure{
    public void affiche(){
        System.out.println("Affichage d'un cercle");
    }
}
interface Affichable{
    void affiche();
}
public class Ex51{
    public static void main(String[] args){
        Affichable[] listeFigures= {new Rectangle(),
                                    new Cercle(),
                                    new Rectangle()} ;
        afficheListe(listeFigures) ;
    }
    static void afficheListe(Affichable [] l){
        for(int i=0 ;i<3 ;i++){
            l[i].affiche();
        }
    }
}
Résultats :
Affichage d'un rectangle
Affichage d'un cercle
Affichage d'un rectangle
```

Commentaire:

On voit l'intérêt de cette approche: toute la classe Ex51 devient indépendante des objets qu'elle manipule, elle ignore le type de ces objets (ce sont des objets de type **Affichable**, ils doivent seulement contenir la méthode **affiche()** .)

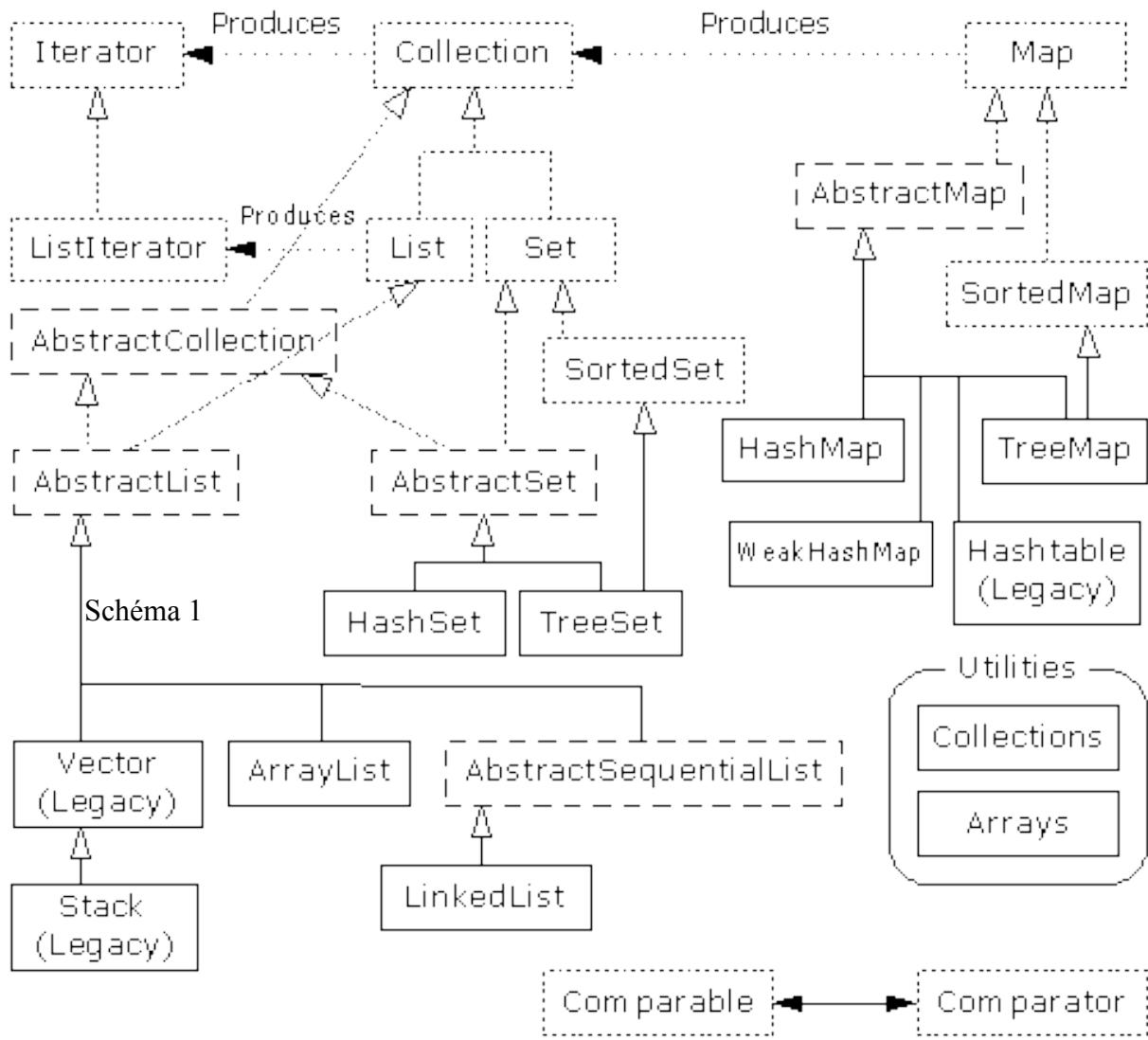
Une interface peut également comprendre des attributs ; dans ce cas ils sont considérés **final** et **static**. Par convention ces attributs s'écrivent en majuscules (dans le cas d'un attribut à plusieurs noms, ces derniers sont alors séparés par des tirets). Toute classe implémentant cette interface pourra utiliser la ou les constantes sans devoir les déclarer.

Ex. 5.2

```
package exemplesPoly.ex52 ;  
  
interface Mois{  
    int NB_JOURS_SEMAINE = 7;  
    int NB_MOIS_ANNEE = 12;  
}
```

6. Les Conteneurs

Les conteneurs désignent un ensemble de classes permettant de gérer de façon approfondie le stockage des objets. Ces conteneurs sont décrits par les interfaces `Collection` et `Map`, elles-mêmes redéfinies en interfaces plus spécifiques comme le montre le schéma 1. Les classes `Vector` et `HashTable` représentaient les seuls outils pour la manipulation d'ensembles dans les versions antérieures à Java 2; leur présence actuelle dans cette hiérarchie assure la compatibilité avec les dernières versions du JDK.



En fait, nous ne traitons dans ce paragraphe que l'interface `Collection` et les classes qui implémentent cette interface: elles sont incluses dans le package `java.util` qu'il sera nécessaire d'importer avant tout usage des fonctionnalités de `Collection`. Nous pourrons donc nous limiter aux objets du schéma 2.

Un conteneur stocke des objets sous forme de références à ces objets, le type est donc perdu. Nous verrons cependant qu'il est possible de le récupérer grâce à l'identification dynamique du type (run time type identification). Les classes implémentant `List` correspondent à des listes: ce sont des suites ordonnées d'éléments quelconques, les doublons sont autorisés. Les

classes implémentant Set correspondent à des ensembles, elles n'acceptent pas les doublons. Pour pouvoir exclure les doublons, toute classe dérivant de Set doit pouvoir comparer les objets qu'elle contient; ces objets doivent donc implémenter l'interface Comparable qui impose de posséder une méthode `int compareTo(Object)`.

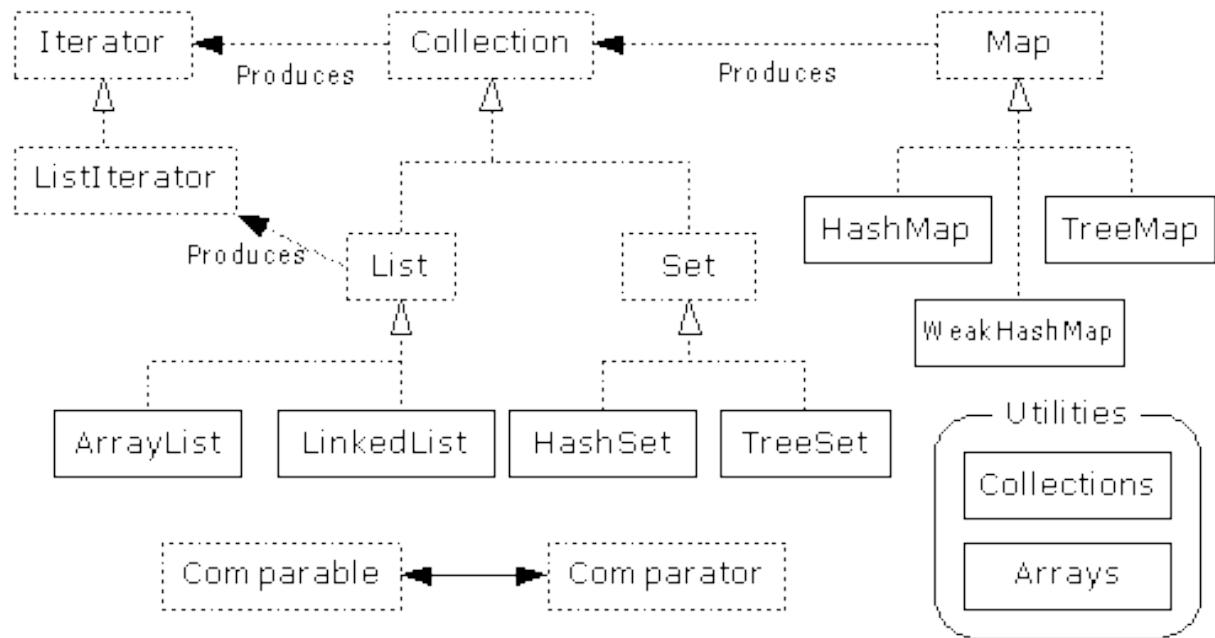


Schéma 2

L'interface Collection possède plusieurs fonctionnalités dont héritent les éléments inférieurs:

<code>boolean add(Object)</code>	ajoute l'argument à la collection
<code>boolean addAll(Collection c)</code>	ajoute tous les éléments de c
<code>void clear()</code>	supprime tous les éléments de la collection
<code>boolean contains(Object o)</code>	indique si l'argument appartient à la collection
<code>boolean containsAll(Collection c)</code>	indique si tous les éléments de c ∈ à la collection
<code>boolean isEmpty()</code>	indique si la collection est vide
<code>Iterator iterator()</code>	renvoie un itérateur de la collection
<code>boolean remove(Object o)</code>	supprime une instance de o de la collection
<code>boolean removeAll(Collection c)</code>	Supprime de la collection tous les éléments de c
<code>int size()</code>	renvoie le nombre d'éléments de la collection
<code>Object[] toArray()</code>	renvoie un tableau des éléments de la collection

L'itérateur renvoyé par la méthode `iterator()` d'une collection permet de parcourir séquentiellement les éléments de la collection.

L'interface Set propose les mêmes fonctionnalités que l'interface Collection. Les Set interdisent les doublons, pour cela la JVM compare tout nouvel élément ajouté à un Set à ceux déjà présents: elle utilise pour cette comparaison la méthode `int compareTo(Object)`

o), cette méthode hérite de la classe `Object` et doit être modifiée par l'utilisateur pour s'adapter au problème traité. Tout objet ajouté à un `Set` doit pouvoir utiliser cette méthode, sa classe devra donc implémenter l'interface `Comparable` (elle contient la méthode `compareTo()`). En cas d'absence de la méthode, la JVM lèvera une `ClassCastException` à l'exécution.

Les `Set` peuvent être instanciés en `TreeSet` ou en `HashSet`. Le choix est conditionné par la taille de l'ensemble à représenter, les fonctionnalités d'un `HashSet` sont plus performantes, par contre le `TreeSet` conserve ses éléments triés.

Les instances de l'interface `List` conservent leurs éléments dans l'ordre d'introduction (elles autorisent les doublons).

Méthodes de l'interface <code>List</code>	Définition
<code>void add(int index, Object element)</code>	insère un objet à l'index spécifié, au sein de la liste courante.
<code>boolean addAll(int index, Coll)</code>	insère tous les éléments de la collection spécifiée, à l'index donné au sein de la liste.
<code>Object get(int index)</code>	retourne l'élément à la position spécifiée dans la liste.
<code>int indexOf(Object o)</code>	retourne la position de la première occurrence de l'objet au sein de la liste courante, ou -1 s'il n'a pu être trouvé.
<code>int lastIndexOf(Object o)</code>	retourne la position de la dernière occurrence de l'objet spécifié au sein de la liste, ou -1 s'il n'a pu être trouvé.
<code>ListIterator listIterator()</code>	retourne un itérateur de liste sur les éléments de l'objet <code>List</code> .
<code>ListIterator listIterator(int index)</code>	retourne un itérateur de liste sur les éléments de l'objet <code>List</code> à partir de la position spécifiée.
<code>Object remove(int index)</code>	supprime l'élément positionné à l'index spécifié au sein de la liste.
<code>Object set(int index, Object element)</code>	remplace l'élément à la position spécifiée dans la liste, par l'objet passé en argument.
<code>List subList(int from, int toI)</code>	retourne une partie de la liste, délimitée par les index <code>from</code> inclus et <code>toI</code> exclus.

Les classes implémentant `List` sont `ArrayList` et `LinkedList`, cette dernière offre des fonctionnalités supplémentaires: `addFirst()`, `addlast()`, `getFirst()`, `getLast()`, `removefirst()` et `removeLast()`. Les deux implémentations se distinguent essentiellement par leurs performances d'accès comme le montre le schéma suivant (les tableaux ont été ajoutés à titre comparatif).

	Lecture	Itération	Insertion	suppression
Tableau	++	+		
<code>ArrayList</code>	+	--	+	--
<code>LinkedList</code>	-	+	++	++

Le parcours séquentiel des éléments d'une Collection est assuré par la classe `Iterator` qui possède les méthodes `boolean hasNext()` et `Object next()`. La méthode `Iterator iterator()` envoyée à une collection renvoie un itérateur auquel on peut envoyer les méthodes `hasNext()` et `next()`.

Ex. 6.1

```
package exemplesPoly.ex61;
import java.util.*;
public class TestCollection{
    public static void main(String[] args){
        Set e = new TreeSet();
        e.add(new Integer(1));
        e.add(new Integer(2));
        e.add(new Integer(3));
        System.out.println("Impr. Collection: "+e);
        Iterator it = e.iterator();
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
}
Résultats:
Impr. Collection: [1, 2, 3]
1
2
3
```

Nécessité d'importer le package java.util.* pour utiliser les Collections.

On peut ajouter tout type d'élément dans une Collection.

Remarque: il n'a pas été utile d'implémenter la méthode `compareTo()` car les classes encapsulant les types primitifs en possèdent une.

Deux instances de Collection peuvent être testées quant à leur égalité grâce à la méthode `boolean equals(Object)` héritée de la classe `Object` pourvu que cette méthode soit définie pour les éléments de la collection (deux collections sont égales au sens de `equals()` si leurs éléments sont égaux au sens de `equals()`). De la même façon, l'envoi de la méthode `String toString()` à une collection provoque l'envoi itéré de cette même méthode aux éléments de la collection; le résultat est une chaîne de caractère démarrant par [et finissant par], les éléments de la collection étant séparés par des virgules. Une mise en œuvre de ces principes est fournie par l'exemple 3.5.1 modifié: le programme principal crée une première liste contenant soit des instances de `VehiculeTourisme` soit des instances de `Utilitaire`. La méthode `int compareTo(Object)` différencie ces instances par leur attribut `numero`. Une deuxième liste est obtenue en reprenant la première liste triée cette fois.

Ex. 6.2

```
package exemplesPoly.ex62;
import java.util.*;
abstract class Transport{
    int numero;
    Transport(int n){numero = n ;}
}
class Vehicule extends Transport implements Comparable{
    String marque ;
    String carburant ;
    Vehicule(int n, String marque){
        super(n) ;
        this.marque = marque ;
    }
    public int compareTo(Object o){
        if (o instanceof Vehicule) return
            compareTo((Vehicule)o);
        return -1;
    }
    int compareTo(Vehicule v){
        return numero - v.numero;
    }
}
```

Les objets ajoutés à une collection doivent pouvoir être comparés

La méthode `compareTo()` doit avoir même signature que dans l'interface comparable

Deux instances de `Vehicule` sont comparées grâce à leur variable `numero`

```

}
class VehiculeTourisme extends Vehicule {
    static int millesime = 2009 ;
    int nbPlaces ;
    VehiculeTourisme(int n, String m){
        super(n,m);
    }
    public String toString(){
        String res = "Véhicule n° " + numero ;
        return res ;
    }
}
class Utilitaire extends Vehicule {
    int volume ;
    Utilitaire(int n, String m){
        super(n,m);
    }
    public String toString(){
        String res = " Utilitaire n° " + numero ;
        return res ;
    }
}

public class Ex62{
    public static void main(String[] args){
        Set ensA = new TreeSet();
        Vehicule v1 = new VehiculeTourisme(45, "Peugeot");
        Vehicule v2 = new Utilitaire(25,"Renault");
        Vehicule v3 = new VehiculeTourisme(35,"Citroen");
        Vehicule v4 = new Utilitaire(35,"Ford");
        ensA.add(v1);
        ensA.add(v2);
        ensA.add(v3);
        ensA.add(v4);
        SortedSet ensB = (SortedSet)ensA;
        System.out.println("Nb. éléments ensA: " +
                           ensA.size());
        System.out.println("Ensemble A: " + ensA);
        System.out.println("Ensemble A trié: " + ensB);
        v2.numero=65;
        System.out.println("Ensemble A trié: " + ensB);
        System.out.println("dernier élément: " +
                           ensB.last());
    }
}

```

Les instances v3 et v4 sont égales au sens de `compareTo()`

L'instance v4 n'est donc pas ajoutée au Set ensA

Même après modification de la variable de tri, l'ordre n'est pas modifié

Le transtypage du Set en SortedSet permet l'accès à des méthodes supplémentaires mais ne change pas l'ordre des instances.

Résultats de l'exécution:
Nb. éléments ensA: 3
Ensemble A: [Utilitaire n° 25, Véhicule n° 35, Véhicule n° 45]
Ensemble A trié: [Utilitaire n° 25, Véhicule n° 35, Véhicule n° 45]
Ensemble A trié: [Utilitaire n° 65, Véhicule n° 35, Véhicule n° 45]
dernier élément: Véhicule n° 45

Commentaires:

On remarquera que les éléments d'un Set sont rangés en fonction du critère (ici la variable `numero`) permettant la comparaison lors de leur ajout; le changement de valeur du critère n'affecte pas l'ordre des éléments (v1 reste le dernier élément). Pour obtenir les éléments triés, il suffit de créer un nouveau SortedSet dans lequel on réintroduit les éléments du premier Set.

La classe Collections:

Cette classe possède plusieurs méthodes `static` permettant de gérer une collection avec en particulier la possibilité d'effectuer des tris. Les tris peuvent être réalisés en utilisant la variable utilisée par la méthode `compareTo()` ou une autre variable en redéfinissant la méthode `compare(Object o1, Object o2)` de l'interface `Comparator`: cette méthode retourne 0 si les objets o1 et o2 sont égaux au sens de l'utilisateur, une valeur différente de 0 sinon.

Les méthodes `compareTo()` et `compare()` diffèrent par leurs usages: `compareTo(Object o)` compare l'objet o à un objet de sa classe tandis que `compare(Object o1, Object o2)` compare deux objets quelconques.

Méthodes (<code>static</code>) de la classe Collections	Définition
<code>int frequency(Collection c, Object o)</code>	retourne le nombre d'objet égaux (au sens <code>equals</code>) à o dans c
<code>boolean disjoint(Collection c1, Collection c2)</code>	retourne true si il n'y a aucun élément commun à c1 et c2
<code>void reverse(List l)</code>	inverse l'ordre des éléments de la liste l
<code>void sort(List l)</code>	trie les éléments de l (ils doivent contenir la méthode <code>compareTo()</code>)
<code>void sort(List l, Comparator c)</code>	trie les éléments de l selon c (ils doivent contenir la méthode <code>compare()</code>)
<code>Object min(Collection c)</code>	retourne le plus petit élément de c (selon <code>compareTo()</code>)
<code>Object min(Collection c, Comparateur cp)</code>	retourne le plus petit élément de c (selon <code>compare()</code>)
<code>Object max(Collection c)</code>	retourne le plus grand élément de c (selon <code>compareTo()</code>)
<code>Object max(Collection c, Comparateur cp)</code>	retourne le plus grand élément de c (selon <code>compare()</code>)

Ex.6.3

```
package exemplesPoly.ex63;
import java.util.*;

public class Ex63 {

    public static void main(String[] args){
        Vehicule v1 = new Vehicule ("Renault Clio",2012,40000);
        Vehicule v2 = new Vehicule ("Peugeot 207",2014,5000);
        Vehicule v3 = new Vehicule ("Opel Corsa",2013,3000);
        List l = new LinkedList();
        l.add(v2);l.add(v1);l.add(v3);
        System.out.println("Classement par ordre d'introduction: \n"+l);

        Collections.sort(l);
        System.out.println("\nClassement par n° \n"+l);

        Vehicule.ComparateurAge cpa = new Vehicule.ComparateurAge();
        Collections.sort(l,cpa);
        System.out.println("\nClassement par année: \n"+l);

        Comparator cpkm = new Comparator(){
            public int compare(Object o1, Object o2){
                return ((Vehicule)o1).km - ((Vehicule)o2).km;
            }
        }; // fin classe anonyme

        Collections.sort(l,cpkm);
        System.out.println("\nClassement par kilométrage: \n"+l);
    }
}
```

Construction d'un comparateur cpk basé sur le kilométrage en instantiant une classe anonyme dérivant de Comparator

```

class Vehicule implements Comparable{
    static int noC = 100;
    int no;
    int annee;
    int km;
    String nom;
    Vehicule(String n,int a, int k){
        annee = a;km = k;no=++noC;nom=n;
    }
    static class ComparatorAge implements Comparator{
        public int compare(Object o1, Object o2){
            return ((Vehicule)o1).annee - ((Vehicule)o2).annee;
        }
    }
    public int compareTo(Object o){
        return no- ((Vehicule)o).no;
    }
    public String toString(){
        return nom+" n° "+no+" année: "+annee+" kilom. "+km;
    }
}

```

Résultats:

Classement par ordre d'introduction:

[Peugeot 207 n° 102 année: 2014 kilom. 5000, Renault Clio n° 101 année: 2012 kilom. 40000, Opel Corsa n° 103 année: 2013 kilom. 3000]

Classement par n°

[Renault Clio n° 101 année: 2012 kilom. 40000, Peugeot 207 n° 102 année: 2014 kilom. 5000, Opel Corsa n° 103 année: 2013 kilom. 3000]

Classement par année:

[Renault Clio n° 101 année: 2012 kilom. 40000, Opel Corsa n° 103 année: 2013 kilom. 3000, Peugeot 207 n° 102 année: 2014 kilom. 5000]

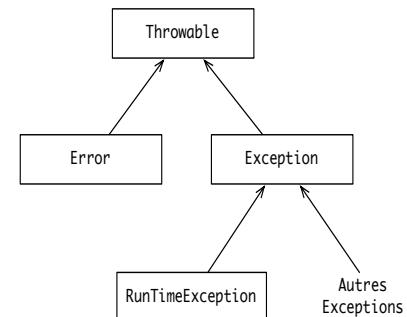
Classement par kilométrage:

[Opel Corsa n° 103 année: 2013 kilom. 3000, Peugeot 207 n° 102 année: 2014 kilom. 5000, Renault Clio n° 101 année: 2012 kilom. 40000]

7. Les Exceptions

La notion d'exception a été introduite en Java pour permettre de séparer le code pouvant générer des erreurs et le traitement de ces erreurs. La classe `Throwable` décrit l'ensemble des exceptions: à toute erreur correspond une instance d'un classe dérivée de `Throwable`. Java classe les exceptions en fonction de leur traitement éventuel par le gestionnaire d'exception:

- La classe `Error` (elle correspond à des erreurs matérielles ou à des erreurs de compilation). La machine virtuelle java gère ces erreurs sans contrôle possible du programmeur.
- La classe `RunTimeException` regroupe les exceptions non contrôlées par le programmeur: elles ne nécessitent pas d'être propagées et capturées (cf. suite du chapitre).
- Les autres exceptions sont laissées sous la responsabilité du programmeur: elles doivent être propagées et capturées.

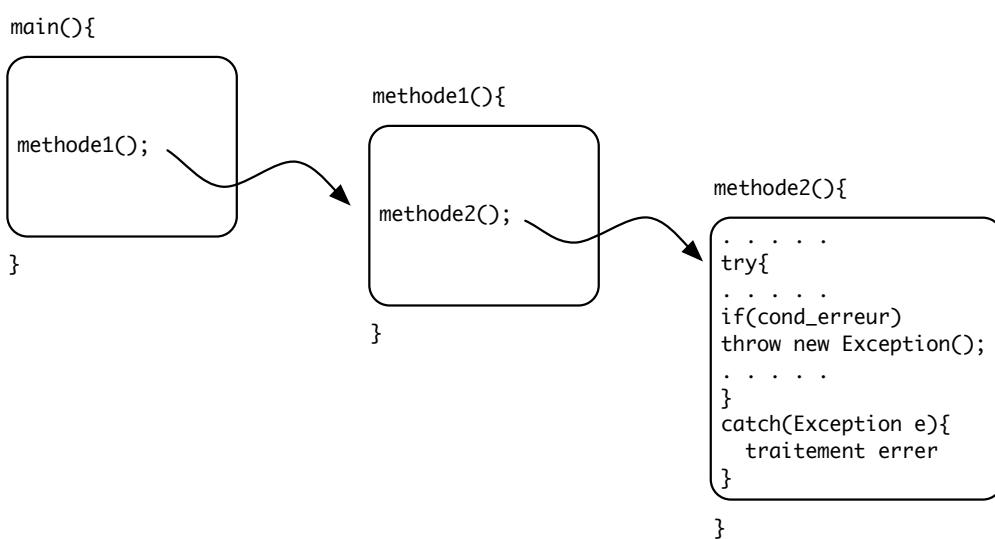


7.1 le mécanisme de base

Le traitement d'une erreur comporte 3 étapes:

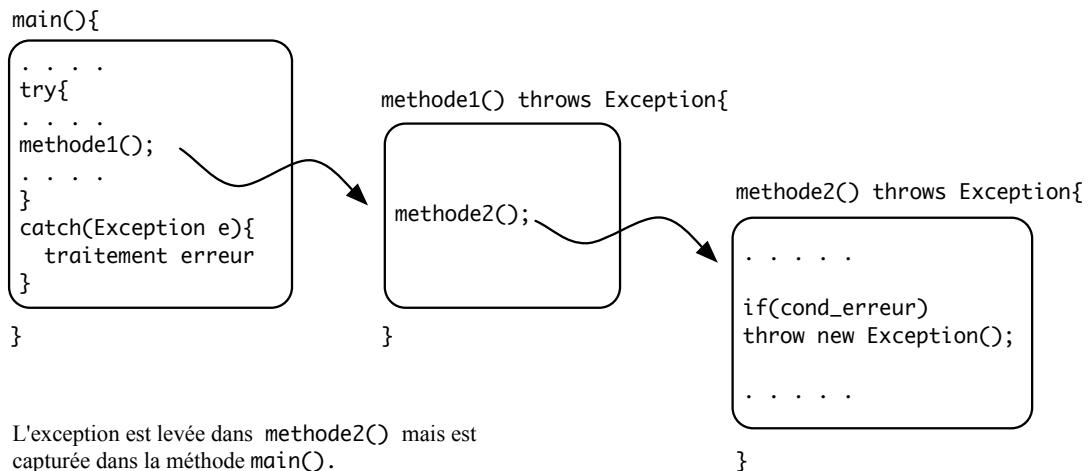
- la détection avec la création d'une instance de la classe `Exception` ou de ses dérivées (mot-clé `throw`)
- la propagation: l'erreur peut être traitée localement ou propagée à (aux) méthode(s) appelante(s) grâce au mot-clé `throws`. Une de ces méthodes doit traiter l'exception, sinon la machine virtuelle Java capture l'exception et interrompt le programme en imprimant le contenu de la pile d'appels (`méthode printStackTrace()`).
- le traitement: il correspond à la capture de l'exception (mots-clé `try{}` `catch{}`)

Le schéma suivant décrit ces appels:



Le schéma ci-dessus correspond à une erreur produite par `methode2()` et corrigée aussitôt grâce au bloc `try{ }catch{ }` . Lorsque la condition menant à l'erreur est vérifiée, le déroulement du bloc `try{ }` est interrompu pour passer au bloc `catch{ }` correspondant au type d'exception levée (plusieurs blocs `catch{ }` correspondant à plusieurs types d'erreur sont possibles), les instructions suivant le bloc `try{ } catch{ }` sont ensuite exécutées.

On peut choisir de propager l'erreur et de la traiter dans le programme principal comme dans le schéma ci-dessous; toutes les exceptions levées sont traitées dans le programme principal. On ne risque pas d'oublier le traitement d'une exception, mais le traitement des exceptions levées devient plus général.



Le bloc `try{} catch{}` peut être complété de façon optionnelle par un bloc `finally{}`: ce dernier est exécuté après les blocs `catch{}` (même si aucune exception n'a été levée).

```

try{
    .
    .
    .
    catch(Exception1 e){ traitement exception }
    catch(Exception2 e){ traitement exception }
    finally{
        .
        .
    }
}

```

le bloc `finally{}` permet d'insérer dans le programme un ensemble d'instructions qui doit être obligatoirement réalisé quelque soit le déroulement du programme (exceptions levées ou non); cela peut concerner par exemple la fermeture d'un fichier.

7.2 Cas particulier: créer ses propres exceptions

Le cas précédent correspondait à l'instanciation de la classe `Exception`; en fonction de l'erreur détectée, il peut être préférable de créer ses propres exceptions en dérivant la classe `Exception` (ou une autre). Les classes dérivant de `Throwable` possèdent presque toutes 2 constructeurs (un constructeur par défaut et un constructeur avec un argument `String` permettant de propager une chaîne de caractères décrivant l'erreur) qu'il faudra redéfinir. Mais la plupart du temps, le choix d'instancier une classe adéquate dérivant de `Exception` permet d'informer correctement l'utilisateur sur la nature de l'erreur (ex. une instance de la classe `NullPointerException` est parfaitement explicite – voir ex. 7.3.1).

7.3 Un exemple

Le programme suivant définit deux classes `VehiculeTourisme` et `VehiculeEco` dérivant d'une classe abstraite `Vehicule` et une classe `Proprietaire`: les instances de `VehiculeEco` doivent toutes avoir un taux de rejet CO2 inférieur à 200 (tâche confiée au constructeur), les valeurs de l'attribut `carburant` sont au nombre de 4 (tâche confiée au constructeur de `VehiculeTourisme`), enfin la méthode `vente()` de `Vehicule` vérifie si l'attribut `prop` a pour valeur une instance de `Proprietaire`.

Ex. 7.3.1

```

package exemplesPoly.ex731;
abstract class Vehicule {
    Proprietaire prop = null;
    int numero;
    String marque ;
    Vehicule(int n, String marque){
        numero = n;
        this.marque = marque ;
    }
    boolean vente() throws Exception{
        int noSerie;
        if(this.prop == null)throw new NullPointerException("Propriétaire
                                                       inconnu");
        return true;
    }
    public String toString(){
        return "véhicule n° " + numero + " " + marque;
    }
}
class Proprietaire{
    String nom;
    String adresse;
}
class VehiculeTourisme extends Vehicule{
    int co2;
    String carburant;
    VehiculeTourisme (int n, String m, String carb) throws Exception{
        super(n,m);
        if(!(carb == "fioul")||(carb == "ess98")||(carb == "gpl"))throw new
            Exception("Erreur carburant: " + carb);
        carburant = carb;
        System.out.println("Fin constructeur VehiculeTourisme");
    }
}
class VehiculeEco extends VehiculeTourisme{
    VehiculeEco(int n, String m, String carb, int co2) throws DepassementCO2,
                                                               Exception{
        super (n,m,carb);
        try{
            if(co2>200)throw new DepassementCO2();
            this.co2 = co2;
        }
        catch (DepassementCO2 e){
            this.co2 = 200;
            System.out.println("Correction co2");
        }
        System.out.println("Fin constructeur VehiculeEco");
    }
}
class DepassementCO2 extends Exception{
//pas de constr. redéfini, donc seul le constr. par défaut est utilisable
}
public class Ex731 {
    public static void main(String[] args) throws Exception{
        try{
            Vehicule v1 = new VehiculeEco(102,"Renault","ess98",230);
            Vehicule v2 = new VehiculeTourisme(101,"Peugeot","fioul");
            Proprietaire p1 = new Proprietaire();
            p1.nom = "Dupont";
            p1.adresse = "Paris";
            System.out.println(v2.vente());
        }
        catch(NullPointerException e){System.out.println(e);}
        System.out.println("Fin main");
    }
}

```

Exception levée et non capturée,
elle est donc propagée

Exception levée et non
capturée, elle est donc
propagée

Exception levée et capturée dans la
même méthode

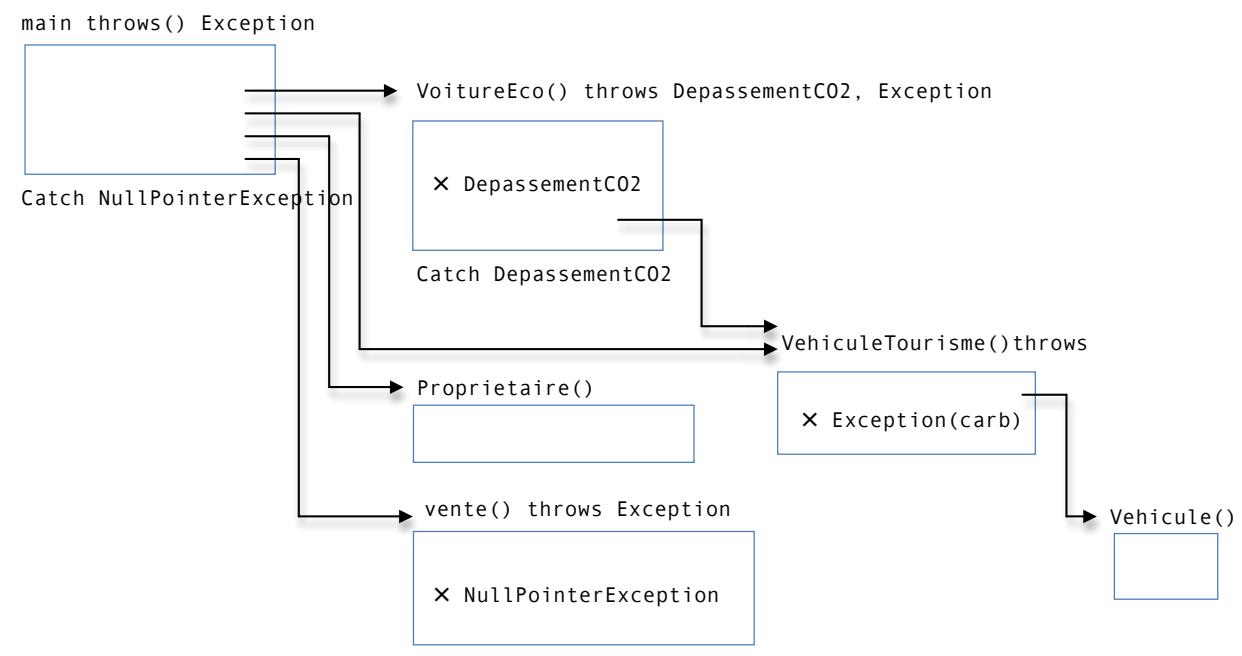
```

Résultats
Fin constructeur VehiculeTourisme
Correction co2
Fin constructeur VehiculeEco
Fin constructeur VehiculeTourisme
java.lang.NullPointerException: Propriétaire inconnu
Fin main
Résultats avec fioul -> fiul
Fin constructeur VehiculeTourisme
Correction co2
Fin constructeur VehiculeEco
Exception in thread "main" java.lang.Exception: Erreur carburant: fiul
at exemplesPoly.ex731.VehiculeTourisme.<init>(Ex731.java:33)
at exemplesPoly.ex731.Ex731.main(Ex731.java:60)

```

Commentaires:

L'exécution du même programme avec une erreur sur le carburant dans l'instanciation de v2 provoque un comportement différent: l'erreur de carburant provoque la levée d'une exception de type Exception dans le constructeur de VehiculeEco, elle est donc propagée mais n'est pas capturée par la méthode main qui ne capture que les exceptions de type NullPointerException. L'exception est finalement prise en charge par la machine virtuelle Java qui affiche le message lié à l'exception et l'état de la pile des appels au moment de la levée de l'exception.



✗ DepassementCO2 ↔ Levée de l'exception DepassementCO2

8. Classes Internes et Interfaces Internes

Une classe interne est une classe déclarée à l'intérieur d'un autre classe (appelée classe externe ou englobante), elle peut être déclarée comme un membre de la classe externe ou à l'intérieur d'une méthode de la classe externe. Cette inclusion est récursive, une classe interne peut contenir des classes internes.

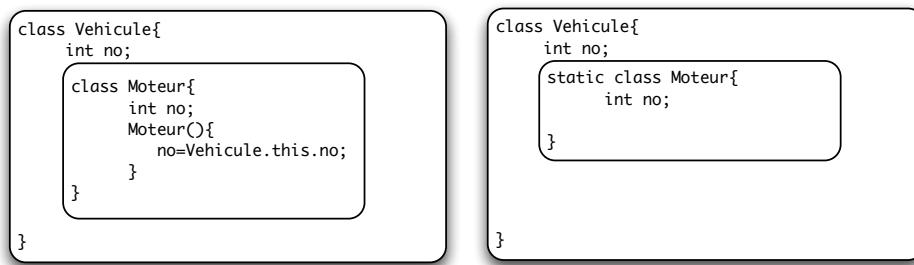
La particularité d'une classe interne est d'offrir une totale visibilité sur les membres de sa classe externe, y compris les membres privés. De même, la classe externe accède aux membres d'une classe interne (y compris les membres privés). On ne peut accéder aux membres d'une classe interne

L'utilisation d'une classe interne peut correspondre à plusieurs besoins:

- exprimer des relations entre classes non prises en compte par héritage (un véhicule comprend un moteur, des accessoires,... mais ces objets n'héritent pas entre eux)
- cacher une partie de l'implémentation: les classes internes peuvent être déclarées privées, elles sont alors invisibles des autres classes y compris celles du package courant.

Les classes internes obéissent à quelques principes de base:

- Des attributs et/ou méthodes de même nom peuvent apparaître dans une classe interne et sa classe englobante. L'ambiguïté est levée grâce au mot-clé `this` (voir exemple).
- Une classe interne peut être considérée comme un membre de sa classe englobante, à ce titre elle a une visibilité et une nature (`static` ou non - dans ce dernier cas, on parlera de classe interne d'instance). Sans autre précision une classe interne doit être considérée comme une classe interne d'instance.
- Cas des classes internes d'instance: **On ne peut accéder aux membres d'une classe interne que par l'intermédiaire de son instance externe**: bien que cela ne soit pas obligatoire, il faut penser à lier l'instance de la classe interne à l'instance de sa classe englobante (dans l'exemple 8.1.1 la classe `Moteur` est une classe interne de la classe `Vehicule`, il faut donc associer une instance de `Vehicule` à une instance de `Moteur` ou réciproquement).
- Cas des classes internes `static`: dans ce cas, les instances d'une classe interne sont créées indépendamment de toute instance de la classe externe. L'instance ainsi créée ne pourra donc pas accéder aux membres non `static` de sa classe englobante mais la classe englobante peut accéder aux membres `static` et non `static` d'une classe interne `static`.



Utilisation

```
Vehicule v1 = new Vehicule();
Vehicule.Moteur m1 = v1.new Moteur();
```

Le constructeur de `Moteur` peut récupérer le `no` de `Vehicule` à la construction de l'instance. Dans ce cas le `no` d'un moteur est celui de son véhicule.

```
Vehicule v2 = new Vehicule();
Vehicule.Moteur m2 = new Vehicule.Moteur();
m2.no = v2.no;
```

Le constructeur de `Moteur` n'a pas accès à la variable d'instance `no` de `Vehicule`. Le `no` doit donc être affecté après.

Les exemples qui suivent mettent en jeu des instances d'une classe `Vehicule`, chaque instance possédant un moteur (instance d'une classe `Moteur`) et une boîte de vitesses (instance d'une classe `BV`). Les classes `Moteur` et `BV` sont tantôt des classes internes, tantôt des classes internes locales ou encore des classes anonymes. Les classes `Vehicule`, `Moteur` et `BV` possèdent toutes un attribut `numero` (la valeur de cet attribut pour `Moteur` et `BV` correspond à la valeur de l'attribut `numero` de `Vehicule`). Les classes `Vehicule` et `Moteur` ont un attribut `carburant` (la valeur de cet attribut pour `Vehicule` correspond à celle de l'attribut `carburant` de `Moteur`).

8.1 Classes internes: exemple d'utilisation

L'exemple suivant reprend le cadre de l'exemple 3.5.1. La description d'un objet de la classe `Vehicule` est complétée par des références sur un moteur et une boîte de vitesses; ces nouveaux objets sont obtenus par instanciation des classes internes `Moteur` et `BV` (private). La construction des instances garantit la correspondance des attributs `numero` des classes `Vehicule`, `Moteur` et `BV` ainsi que la correspondance de l'attribut `carburant` pour les classes `Vehicule` et `Moteur` (cf. Fig. ci-dessus). La classe `Vehicule` possède deux constructeurs, selon que le type de boîte est ou n'est pas précisé.

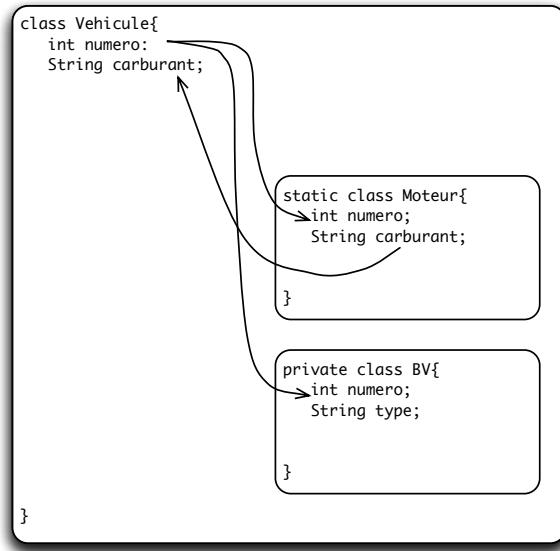
Dans un premier temps l'instance `v1` est construite en précisant le type de boîte et en appelant la méthode `motorisation()`. Puis une tentative de construire une instance `v2` identique à la précédente (au numéro près) est produite en initialisant un à un les attributs. Cette tentative échoue car la classe `BV` étant inaccessible, il faut affecter à `v2` une boîte par défaut grâce à la méthode `transmissionDefaut()`.

Ex. 8.1.1

```
package exemplesPoly.ex811;

abstract class Transport{
    int numero;
    static int numeroCourant = 100;
    Transport(){numero = ++numeroCourant;}
}
class Vehicule extends Transport{
    String marque ;
    String carburant ;
    Moteur moteur;
    BV bv;
    Vehicule(String marque){
        super() ;
        this.marque = marque ;
    }
    Vehicule(String marque, String typeBV){
        super();
        this.marque = marque ;
        bv=new BV(typeBV);
    }
    void motorisation(String carburant){
```

Les attributs `moteur` et `bv` sont essentiels: ils permettent d'associer les instances de la classe interne à la classe externe.



```

        moteur = new Moteur(carburant);
        moteur.numero = numero;
        this.carburant = carburant;
    }
    void transmissionDefaut(){
        bv=new BV("man4v");
    }
    static class Moteur{
        String carburant;
        int numero;
        Moteur(String carburant){
            this.carburant = carburant;
        }
        Moteur(){
        }
        public String toString(){
            return "n° "+numero+" carb. "+carburant;
        }
    }
    private class BV{
        String type;
        int numero;
        BV(String type){
            this.type=type;
            numero=Vehicule.this.numero;
        }
        public String toString(){return numero+" "+type;}
    }
    public String toString(){
        return "Vehicule "+marque+" n° "+numero+" carb.
            "+carburant+" moteur "+moteur+" BV "+bv;
    }
}
public class Ex811{
    public static void main(String[] args){
        Vehicule v1 = new Vehicule( "Peugeot", "auto5v" ) ;
        v1.motorisation("essence");

        // Tentative de Construction d'une instance équivalente
        // (au n° près)
        // en utilisant le constr. Vehicule(marque)
        Vehicule.Moteur m2 = new Vehicule.Moteur();
        Vehicule v2 = new Vehicule("Peugeot");
        m2.carburant = "essence";
        m2.numero = v2.numero;
        v2.moteur = m2;
        v2.carburant = "essence";
        // Vehicule.BV bv2 = v2.new BV();
        v2.transmissionDefaut();

        System.out.println(v1) ;
        System.out.println(v2) ;
    }
}

/* Résultats:
Vehicule Peugeot n° 101 carb. essence moteur n° 101 carb. essence BV 101 auto5v
Vehicule Peugeot n° 102 carb. essence moteur n° 102 carb. essence BV 102 man4v
*/

```

Le constructeur de cette classe static ne peut avoir accès aux attributs non static de la classe englobante.
L'initialisation se fait dans la méthode motorisation()...

Le constructeur assure la correspondance du numero entre l'instance interne et l'instance externe.
"this" permet de résoudre le problème d'homonymie d'attribut.

D'ici la classe BV est inaccessible.

Commentaires: le lien entre une instance de classe interne et l'instance de sa classe englobante n'existe que pour garantir l'accès réciproque entre les membres de l'instance interne et les membres de l'instance externe (visibilité totale entre membres). Cet accès est utilisé naturellement par les

constructeurs des classes concernées. Dans le cas de l'instance v2 construite avec les constructeurs par défaut, il faut préciser à chaque étape l'instance (interne ou externe) concernée.

8.2 Classes internes locales

L'utilisation d'une classe interne locale (i.e. une classe interne déclarée à l'intérieur d'un bloc, typiquement une méthode) permet d'augmenter la protection du code, de le rendre plus robuste ou de pallier l'absence de "closure" en Java. En effet le code est inaccessible aux autres classes et même à la classe contenant la méthode (hormis la méthode contenant la classe interne). Les règles de visibilité changent par rapport aux classes internes précédentes: une classe définie dans un bloc de code n'est pas un membre de la classe englobante, n'étant pas accessible en dehors du bloc où elle est définie sa visibilité (public, private, protected, package) et sa nature (static ou non) n'ont pas de sens et ne peuvent donc être utilisées. Sa visibilité est toujours limité au bloc qui la contient. D'autre part, si la classe interne locale utilise les variables de sa méthode ou les paramètres de sa méthode, ceux-ci devront avoir été déclarés final.

L'exemple 8.2.2 dérivé du précédent illustre ces possibilités: l'ajout d'un moteur et d'une boîte de vitesses se fait par l'appel des méthodes `motorisation(carburant String)` et `transmission(String type)`. Ces méthodes créent une instance de la classe `Moteur` et une instance de la classe `BV` (ces classes étant déclarées dans leur corps de méthode respectif).

Ex. 8.2.1

```
package exemplesPoly.ex821;
import java.util.*;
abstract class Transport{
    int numero;
    static int numeroCourant = 100;
    Transport(){numero = ++numeroCourant;}
}
class MoteurType{
    int numero;
    String carburant;
    public String toString(){
        return "n° "+numero+" carb. "+carburant;
    }
}
class Vehicule extends Transport{
    String marque ;
    String carburant ;
    MoteurType moteur;
    Object bv;
    Vehicule(String marque){
        super() ;
        this.marque = marque ;
    }
    void motorisation(final String carbu){
        class Moteur extends MoteurType{
            Moteur(){
                numero = Vehicule.this.numero;
                this.carburant = carbu;
                Vehicule.this.carburant = carbu;
            }
        }
        moteur = new Moteur();
    }
    void transmission(String type){
        final String typeT = type;
        class BV{
            int numero;
            String type;
        }
    }
}
```

La variable `carbu` est utilisée dans la classe interne locale: elle doit donc être déclarée final.

Une classe interne peut dériver d'une classe quelconque.

`typeT` est déclaré final pour pouvoir être utilisé dans la classe interne `BV`.

```

        BV(String type){
            numero = Vehicule.this.numero;
            this.type = typeT;
        }
        public String toString(){return numero+" "+type;}
    }
    bv = new BV(type);
}
public String toString(){
    return "Vehicule "+marque+" n° "+numero+ " carb.
        "+carburant+ " moteur "+moteur+ " BV "+bv;
}
}

public class Ex821 {
    public static void main(String[] args){
        Vehicule v1 = new Vehicule("Peugeot") ;
        v1.motorisation("essence");
        v1.transmission("auto5v");
        System.out.println(v1) ;
    }
}

/* résultats
Vehicule Peugeot n° 101 carb. essence moteur n° 101 carb. essence BV 101 auto5v
*/

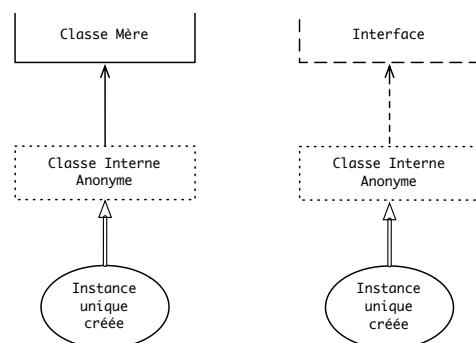
```

Commentaires:

En dehors des méthodes les classes internes locales ne sont pas visibles: l'attribut moteur de la classe Vehicule ne peut donc pas être de type Moteur mais il peut être de type MoteurType (qui est visible et dont la classe Moteur hérite); ce n'est pas le cas pour la classe BV, l'attribut bv est donc de type Object.

8.3 Classes internes anonymes

Elles constituent un cas particulier des classes internes: lorsqu'une seule instance est nécessaire, sa classe devient inutile. Cette unique instance est alors produite à partir d'une classe sans nom (anonyme) qui est soit la dérivée d'une classe mère soit l'implémentation d'une interface. En résumé, une seule instruction créera la classe (anonyme) et son (unique) instance. L'avantage est d'améliorer la lisibilité du code puisque classe et instance sont proches, de plus il n'y a pas de nom de classe à gérer. Une classe anonyme redéfinit (en général) les méthodes de sa classe mère et peut contenir de nouvelles méthodes, mais ces nouvelles méthodes ne seront pas utilisables de l'extérieur car de l'extérieur l'instance est vue comme une instance de la classe mère (qui ne contient pas les nouvelles méthodes). Pour la même raison, il est possible de définir des attributs dans une classe locale, mais ils seront invisibles de l'extérieur. La visibilité des membres de la méthode englobante est identique à celle des classes internes locales: les arguments de méthode doivent être déclarés final (dans le cas où l'argument doit être modifié, il faut l'incorporer dans un tableau déclaré final).

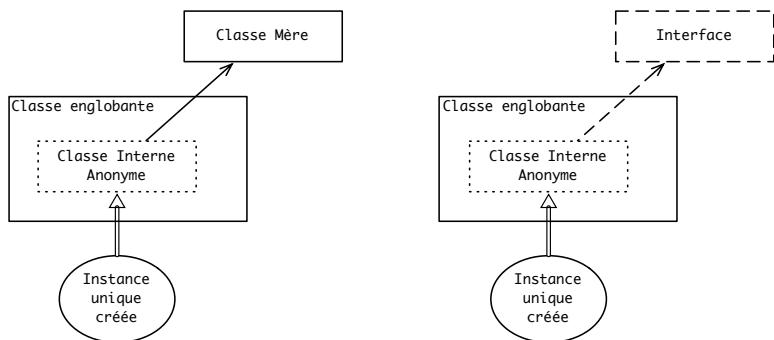


Une utilisation typique est l'implémentation d'un Listener (une seule instance surveille le clavier) ou d'un thread unique par implémentation de l'interface Runnable (cf. Cours Java Avancé).

N'ayant pas de nom, une classe anonyme ne possède pas de constructeur. **La définition de la classe et son instantiation correspondent à une seule expression, sa définition se termine donc par ";"**. La création d'une classe anonyme correspond à l'une des formes syntaxiques suivantes:

```
new ClasseMere(param) {corps classe anonyme} ;
ou si la classe est créée par appel de méthode
methode() {...new ClasseMere(param) {corps classe anonyme} ;...}
```

Comme l'indique le schéma ci-dessous, une instance (unique) est créée à partir de la classe anonyme elle-même dérivée de Classe Mère (cette dernière pouvant être extérieure à la classe englobante).



L'exemple 8.3.1 reprend l'exemple 8.2.1 en modifiant l'association moteur et boîte: la méthode motorisation (final String carb) crée une instance de Moteur en instanciant une classe anonyme dérivée de la classe Moteur qui est hors de la classe Vehicule. Si le constructeur Vehicule(String marque, String type) est utilisé il associe au véhicule une boîte instanciant une classe anonyme implémentant l'interface BV, la classe anonyme redéfinit les "getters" getNumero() et gettype(). Ces "getters" sont nécessaires car une classe anonyme implémentant une interface ne possède pas d'attributs hérités, si des attributs lui étaient ajoutés ils ne seraient pas accessibles de l'extérieur. Si le constructeur Vehicule(String marque) est utilisé, il appelle la méthode bvDefaut() qui de la même façon crée une boîte par défaut (man4v) en instanciant une classe anonyme implémentant l'interface BV.

Ex. 8.3.1

```
package exemplesPoly.ex831;
abstract class Transport{
    int numero;
    static int numeroCourant = 100;
    Transport(){numero = ++numeroCourant;}
}
class Moteur {
    String carburant;
    int numero;
    public String toString(){
        return "n° "+getNumero()+" carb. "+carburant;
    }
    int getNumero(){return 0;}
    String getCarburant(){return "to";}
}
```

```

        Moteur(String carburant){
            this.carburant = carburant;
        }
    }
    interface BV{
        String getType();
        int getNumero();
    }
    class Vehicule extends Transport{
        String marque ;
        String carburant ;
        Moteur moteur;
        BV bv;
        BV bvDefaut(){
            return new BV(){
                int numero = Vehicule.this.numero;
                public String getType(){return "man4v";}
                public int getNumero(){return Vehicule.this.numero;}
                public String toString(){return numero+" "+getType();}
            }; // fin classe anonyme
            // on vérifie la boîte créée
            System.out.println(bv);
        }
        Vehicule(){
        }
        Vehicule(String marque){
            super() ;
            this.marque = marque ;
            bv=bvDefaut();
        }
        Vehicule(String marque, final String typeBV){
            super();
            this.marque = marque ;
            bv=new BV(){
                int numero = Vehicule.this.numero;
                public String getType(){return typeBV;}
                public int getNumero(){return Vehicule.this.numero;}
                public String toString(){return numero+" "+getType();}
            }; // fin classe anonyme
            // on vérifie la boîte créée
            System.out.println(bv);
        }
        void motorisation( final String carb){
            carburant = carb;
            moteur = new Moteur(carb){
                int getNumero(){return Vehicule.this.numero;}
                String getCarburant(){return carb;}
            }; // fin classe anonyme
        }
        public String toString(){
            return "Vehicule "+marque+" n° "+numero+ " carb. "+
            carburant+ " moteur "+moteur+ " BV "+bv;
        }
    }
    public class Ex831{
        public static void main(String[] args){
            Vehicule v1 = new Vehicule( "Peugeot","auto5v") ;
            v1.motorisation("hybride");
            System.out.println(v1) ;
            Vehicule v2 = new Vehicule("Renault");
            v2.motorisation("essence");
            System.out.println(v2) ;
        }
    }

```

Les attributs d'une classe anonyme ne peuvent être vus de l'extérieur, il faut donc redéfinir les "getters" de l'interface pour accéder à ces informations.

L'argument typeBV doit être déclaré final pour pouvoir être utilisé dans le contenu de la classe anonyme.

```

}

/*
Résultats
101 auto5v
Vehicule Peugeot n° 101 carb. hybride moteur n° 101 carb. hybride BV 101 auto5v
102 man4v
Vehicule Renault n° 102 carb. essence moteur n° 102 carb. essence BV 102 man4v
*/

```

Commentaire: aussitôt instanciée, la boîte est affichée pour vérification.

8.4 Classes internes et Interfaces imbriquées

De la même façon que les classes internes, les interfaces peuvent être imbriquées. A la différence des interfaces standard qui ont la visibilité `public` ou `package`, une interface interne peut avoir la visibilité `private`. Dans ce cas, si une classe implémente cette interface, ses instances ne pourront pas être transtypées vers l'interface. La nature d'une interface interne est toujours `static` (ce qui n'implique pas que la classe implémentant une interface interne soit `static`). De plus classes internes et interfaces internes peuvent être mélangées: une classe peut contenir une interface et une interface peut contenir une classe.

L'exemple 8.4.1 reprend les exemples précédents en ajoutant des interfaces. L'interface `Vehiculable` comprend une interface interne `BVable` ET une classe interne `BV_Defaut` (toutes deux de nature `static`). La classe `Vehicule` comprend une classe interne `BV` et une classe interne `Moteur` implémentant une interface `Motorisable` contenue dans la même classe englobante. Il est à noter que l'implémentation d'une interface interne par une classe interne n'est pas obligatoire (bien que `BVable` soit une interface interne de `Vehiculable`, son implémentation `Vehicule` pourrait ne pas contenir de classe interne implémentant `BVable`).

Ex. 8.4.1

```

package exemplesPoly.ex841;

abstract class Transport{
    int numero;
    static int numeroCourant = 100;
    Transport(){numero = ++numeroCourant;}
}
interface Vehiculable{
    int getNo();
    void motorisation(String carburant);
    interface BVable{
        String getType();
        int getNo();
    }
    class BV_defaut implements BVable{
        int numero;
        BV_defaut(int no){numero = no;}
        public int getNo(){return numero;}
        public String getType(){return "man4v";}
        public String toString(){return getNo()+" "+getType();}
    }
}
// une interface interne est static: BVable est implementable
// sans nécessité d'implémenter Vehiculable
// mais ne signifie pas que l'implémentation de BVable soit une
// classe interne static

```

Une interface peut contenir une classe (obligatoirement de nature static).

```

class Vehicule extends Transport implements Vehiculable{
    String marque;
    String carburant;
    Motorisable moteur;
    BVable bv;
    static int noC=10;
    public int getNo(){return numero;}

    public void motorisation(String carb){
        moteur = new Moteur(carb);
    }

    interface Motorisable{
        int getNumero();
        String getCarburant();
    }
    class Moteur implements Motorisable{
        int numero;
        String carburant;
        public int getNumero(){return numero;}
        public String getCarburant(){return carburant;}
        public String toString(){
            return "n° "+numero+" carb. "+carburant;
        }
        Moteur(){
            numero = Vehicule.this.numero;
            Vehicule.this.carburant = "diesel";
            carburant = "diesel";
        }
        Moteur(String carb){
            numero = Vehicule.this.numero;
            Vehicule.this.carburant = carb;
            carburant = carb;
        }
    }
    class BV implements BVable{
        String typeBV;
        public int getNo(){return numero;}
        public String toString(){return getNo()+" "+getType();}
        public String getType(){return typeBV;}
        BV(String typeBV){
            this.typeBV = typeBV;
        }
    }
    Vehicule(String marque){
        super();
        this.marque = marque;
        // le numero du Vehic. doit être transmis au constructeur
        // car aucun moyen de le récupérer dans
        // l'interface Vehiculable
        bv = new BV_defaut(numero);
    }
    Vehicule(String marque, String typeBV){
        this(marque);
        bv = new BV(typeBV);
    }
    public String toString(){
        return "Vehicule "+marque+" n° "+numero+" carb. "+
        carburant+" moteur "+moteur+" BV "+bv;
    }
}

```

Une classe peut contenir une interface.

BVable est une interface interne static. Elle est tantôt implémentée par une classe static (car BV_defaut est une classe static) tantôt implémentée comme ici par une classe interne d'instance (car BV est une classe interne d'instance).

Le numero est transmis car la classe BV_defaut ne peut accéder à l'attribut numero qui est inexistant dans l'interface englobante.

```

public class Ex841 {

    public static void main(String[] args) {
        Vehicule v1 = new Vehicule("Peugeot","auto5v");
        v1.motorisation("essence");
        System.out.println(v1);
        Vehicule v2 = new Vehicule("Renault");
        v2.motorisation("diesel");
        System.out.println(v2);
        Vehiculable.BVable bv1 = new Vehiculable.BVdefaut(111);
        Vehiculable.Bvable bv2 = v1.new BV("man6v");
        System.out.println(bv1);
        System.out.println(bv2);
    }
}

/*
Vehicule Peugeot n° 101 carb. essence moteur n° 101 carb. essence BV 101 auto5v
Vehicule Renault n° 102 carb. diesel moteur n° 102 carb. diesel BV 102 man4v
111 man4v
101 man6v
*/

```

Commentaire:

L'instanciation d'une boîte par défaut crée une boîte de type "man4v".

Si une interface ne peut pas être instanciée elle peut servir de prototype à une classe anonyme et permettre une instanciation comme le montre l'exemple suivant: la méthode **main** de la classe **Ex842** crée une instance de l'interface **Motorisable** en une seule instruction (redéfinition des méthodes de l'interface et instanciation).

Ex. 8.4.1

```

package exemplesPoly.ex842;

interface Motorisable{
    int getNo();
    String getMotorisation();
}

public class Ex842 {
    static int noC = 100;
    public static void main(String[] arg){

        Motorisable m1 = new Motorisable(){
            public int getNo(){return ++noC;}
            public String getMotorisation(){return "diesel";}
            public String toString(){return "Véhicule n° "+getNo()+
                " motorisation: "+getMotorisation();}
        }; //fin classe anonyme

        System.out.println(m1);
    }
}

Résutat:
Véhicule n° 101 motorisation: diesel

```

9. Compléments

9.1 Java et la Réflexion

La réflexion (ou introspection) est un ensemble d'outils Java permettant d'obtenir dynamiquement des informations sur les éléments d'un programme en cours d'exécution (nom des classes, constructeurs, membres, ...). Ces informations, encore appelées méta-données, sont stockées dans la classe `Class`. C'est une classe `final`, à chaque classe créée dans le programme correspond une instance de cette classe matérialisée par un fichier portant l'extension `.class`. Le tableau suivant récapitule ses méthodes:

Méthodes de Class	
<code>newInstance()</code>	crée une nouvelle instance du type de l'objet
<code>String getName()</code>	retourne le nom complet de la classe
<code>String getSimpleName()</code>	retourne le nom de la classe
<code>Class getSuperClass()</code>	retourne la classe mère
<code>getPackage()</code>	
<code>getInterfaces()</code>	
<code>getConstructors()</code>	
<code>Field[] getDeclaredFields()</code>	retourne tous les attributs décl. dans la classe
<code>Field[] getFields()</code>	retourne les seuls attributs <code>public</code>
<code>Method[] getDeclaredMethods()</code>	retourne les méthodes décl. dans la classe
<code>Class forName(String n)</code>	retourne un objet <code>Class</code> où <code>n</code> est un nom abs.
<code>o instanceof classe</code>	vrai si <code>o</code> ∈ <code>classe</code> direct. ou par héritage
Méthodes de Field	
<code>String getName()</code>	retourne le nom de l'attribut
<code>Class getType()</code>	retourne le type de l'attribut
<code>Object get(o)</code>	retourne la valeur de l'attr. pour l'objet <code>o</code>
<code>int getInt(o)</code>	retourne la valeur entière de l'attr. pour l'objet <code>o</code>
<code>float getFloat(o)</code>	retourne la valeur flot. de l'attr. pour l'objet <code>o</code>
Méthodes de Method	
<code>String getName()</code>	retourne le nom de la méthode
<code>Class getReturnType()</code>	Retourne le type retour de la méthode
<code>Object invoke(o1,o2, . . .)</code>	Applique la méth. Sur <code>o1</code> avec les param. <code>o2,...</code>

Le polymorphisme est un exemple de mise en œuvre de ces outils: il permet de déterminer la méthode à appliquer à l'objet au moment de l'exécution. L'exemple suivant montre l'utilisation de la réflexivité; le tableau `tab[]` contient une liste de noms de classes, le programme principal crée ensuite une instance de chaque classe et la place dans la `LinkedList` `liste`. En connaissant uniquement le nom de la classe, le programme instancie cette classe, les constructeurs respectifs affichent alors le résultat. Pour cela on utilise la méthode `forName(n)` de la classe `Class`, la méthode retourne une instance de `Class` dont le nom est `n` (en effet à toute classe correspond une instance de `Class`). La méthode `forName(n)` rend obligatoire la gestion d'une exception en cas de classe inexistante (ici, l'élément correspondant à la classe erronée est sauté).

Ex. 9.1.1

```
package exemplesPoly.ex911;
import java.util.*;
class Cercle{
    Cercle(){System.out.println("Création d'un cercle!");}
```

```

class Rectangle{
    Rectangle(){System.out.println("Création d'un
                                    rectangle!");}
}
class Losange{
    Losange(){System.out.println("Création d'un losange!");}
}
public class Ex91 {
    static String tab[] = {"Rectangle", "ercle", "Cercle"};
    static List liste = new LinkedList();
    public static void main (String[] args) throws Exception{
        for(int i=0;i<tab.length;i++){
            try{
                liste.add(Class.forName("exemplesPoly.ex91." +
                                         tab[i]).newInstance());
            }
            catch(ClassNotFoundException e){
                System.err.println("Classe inexistante");
                continue;
            }
        }
    }
}

Résultats
Création d'un rectangle!
Classe inexistante
Création d'un cercle!

```

L'exemple suivant est plus complexe et montre toutes les possibilités de la réflexivité: il s'agit de trouver l'élément maximum d'une collection d'objets à partir du nom de l'attribut utilisé pour la comparaison. On supposera que cet attribut est de type primitif. Pour chaque objet v de la collection, on réalise les opérations suivantes:

- récupération de la classe de v grâce à `getClass()`
- récursion des attributs de la classe grâce à `getDeclaredFields()` (sous forme de tableau de type `Field[]`)
- parcours du tableau précédent à la recherche d'un attribut correspondant au nom cherché (méthode `getName()`)
- récupération de la valeur de cet attribut grâce à la méthode `get(v)`

Pour tenir compte des différents types primitifs possible (méthode `getType()`), la même opération est effectuée pour chaque type primitif.

Ex. 9.1.2

```

package exemplesPoly.ex912;
import java.lang.reflect.Field;
import java.util.*;

public class Ex92 {
    public static void main(String[] args){
        Vehicule v1 = new Vehicule ("Renault Clio",2012,40000,1.4f);
        Vehicule v2 = new Vehicule ("Peugeot 207",2014,5000,1.1f);
        Vehicule v3 = new Vehicule ("Opel Corsa",2013,50000,1.2f);
        List l = new LinkedList();
        l.add(v2);l.add(v1);l.add(v3);
        System.out.println("Liste des Véhicules: \n"+l);
        System.out.println("Véhicule ayant le kilométrage le plus élevé:
                           "+Vehicule.max(l, "km"));
        System.out.println("Véhicule ayant le poids le plus élevé:
                           "+Vehicule.max(l, "poids"));
    }
}

```

```

class Vehicule{
    static int noC = 100;
    int no;
    int annee;
    int km;
    float poids;
    String nom;
    Vehicule(String n,int a, int k,float p){
        annee = a;km = k;no=++noC;nom=n;poids=p;
    }
    public String toString(){
        return nom+" n° "+no+" année: "+annee+" kilom. "+km+
               " poids: "+poids;
    }
    public static Vehicule max(Collection l,String attribut){
        Vehicule v= null;
        Vehicule objetMax = null;
        Object valMax = null;
        Class c;
        boolean attributExiste = false;
        Iterator it = l.iterator();
        while(it.hasNext()){
            v = (Vehicule)(it.next());
            c = v.getClass();
            Field[] tab = c.getDeclaredFields();
            for(int i=0;i<tab.length;i++){
                try{
                    if(tab[i].getName().equals(attribut)){
                        attributExiste = true;
                        Class t = tab[i].getType();
                        if(tab[i].getType().toString().equals("int")){
                            int n = tab[i].getInt(v);
                            if(valMax == null)valMax=0;
                            if(n > (Integer)valMax){
                                objetMax = v;
                                valMax = n;
                            }
                        }
                    }
                    if(tab[i].getName().equals(attribut)){
                        attributExiste = true;
                        Class t = tab[i].getType();
                        if(tab[i].getType().toString().equals("float")){
                            float f = tab[i].getFloat(v);
                            if(valMax == null)valMax=0f;
                            if(f > (Float)valMax){
                                objetMax = v;
                                valMax = f;
                            }
                        }
                    }
                    if(tab[i].getName().equals(attribut)){
                        attributExiste = true;
                        Class t = tab[i].getType();
                        if(tab[i].getType().toString().equals("double")){
                            double d = tab[i].getDouble(v);
                            if(valMax == null)valMax=0d;
                            if(d > (Double)valMax){
                                objetMax = v;
                                valMax = d;
                            }
                        }
                    }
                } catch (IllegalArgumentException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                }
            }
            if(!attributExiste)System.out.println("l'attribut "+attribut+
                                               " n'existe pas");
            return objetMax;
        }
    }
}

```

Pour chaque objet de la liste, on parcourt la liste de ses attributs

Cas d'un attribut de type int

Cas d'un attribut de type float

Cas d'un attribut de type double

Résultats:

Liste des Véhicules:

[Peugeot 207 n° 102 année: 2014 kilkom. 5000 poids: 1.1, Renault Clio n° 101 année: 2012 kilkom. 40000 poids: 1.4, Opel Corsa n° 103 année: 2013 kilkom. 50000 poids: 1.2]
Véhicule ayant le kilométrage le plus élevé : Opel Corsa n° 103 année: 2013 kilkom. 50000 poids: 1.2
Véhicule ayant le poids le plus élevé : Renault Clio n° 101 année: 2012 kilkom. 40000 poids: 1.4

9.2 Le traitement des dates

La classe `Calendar` est une classe abstraite implémentable par une unique classe, `GregorianCalendar`. Elle propose un ensemble de méthodes et de champs i.e. des attributs `static (YEAR,MONTH,DAY_OF_MONTH,HOUR,WEEK_OF_MONTH,...)` permettant de travailler sur des dates. Une instance de `Calendar` est donc une date et un ensemble d'outils pour la manipuler. La classe `Date` permet de représenter un instant donné avec une précision d'une milliseconde. La classe `TimeZone` permet de manipuler un fuseau horaire à partir du méridien de Greenwich. Le tableau suivant fournit des méthodes permettant de créer une date et de la manipuler.

<code>Calendar getInstance()</code>	crée une date de type <code>Calendar</code> à l'aide d'une classe concrète implémentant toutes les méthodes abstraites de <code>Calendar</code> (calendrier Grégorien par défaut)
<code>GregorianCalendar(int a,int m,int d,int h,int m,int s)</code>	crée une date de type <code>Calendar</code> : a: année, m: mois, d: jour, ...
<code>Date getTime()</code>	retourne un objet <code>Date</code> exprimant le tps de l'évènement
<code>long getTimesInMillis()</code>	Retourne la date de l'évènement exprimée en ms à partir du 1 ^{er} Janvier 1970
<code>int get(int champ)</code>	récupère un élément de la date à partir du champ spécifié (ex. <code>MONTH</code> , <code>DAY_OF_WEEK</code>)
<code>void set(int champ, int val)</code>	modifie le champ spécifié
<code>void add(int champ, int val)</code>	augmente le champ de la valeur indiquée avec respect des règles calendaires
<code>boolean isLenient()</code>	indique si la date est correcte
<code>boolean isLeapYear(int a)</code>	indique si a est une année est bissextile
<code>int getMinimalDaysInFirstWeek()</code>	retourne le nb de jours de la 1 ^{ère} semaine

Ces méthodes utilisent un argument `champ` qui est une valeur entière prédéfinie (`static`); les champs utilisables peuvent être pris parmi les suivants:

<code>HOUR_OF_DAY</code>	L'heure (0 à 23)
<code>MINUTE</code>	Le nombre de minutes (0 à 59)
<code>DAY_OF_WEEK</code>	Le jour de la semaine (0 à 6)
<code>DAY_OF_MONTH</code>	Le jour du mois (1 à 31)
<code>MONTH</code>	Le numéro du mois (0 à 11)
<code>WEEK_OF_MONTH</code>	Le numéro de la semaine dans le mois
<code>WEEK_OF_YEAR</code>	Le n° de la semaine dans l'année
<code>YEAR</code>	L'année

Le programme suivant illustre l'utilisation de ces outils (l'import du package `java.util.*` est nécessaire):

Ex. 9.2.1

```
package exemplesPoly.ex921;

import java.util.*;
public class TestCalendar {
    public static void main(String[] args){
        final long CONST_DURATION_OF_DAY = 1000 * 60 * 60 * 24;
        Calendar d0 = new GregorianCalendar();
        Calendar d0bis = Calendar.getInstance();

        Calendar d1 = new GregorianCalendar(2015,2,14,17,65,3);

        // modifications et impression
        System.out.println("Date d0 \t\t\t" + d0.getTime());
        System.out.println("Date d0bis \t\t\t" + d0bis.getTime());
        d0bis.set(Calendar.MONTH, 11);
        System.out.println("Date d0bis_modifiée \t\t\t" + d0bis.getTime());
        d0bis.setTime(new Date());
        System.out.println("Date d0bis_rétablissement \t\t\t" + d0bis.getTime());
        int j = d0.get(Calendar.DAY_OF_MONTH);
        int m = d0.get(Calendar.MONTH);
        int y = d0.get(Calendar.YEAR);
        int d = d0.get(Calendar.DAY_OF_YEAR);
        System.out.println("le " + j + " du " + (m + 1) + " ème mois de l'année
                           " + y + " soit le " + d + " ème jour de l'année " + y+"\n");

        System.out.println("Date d1 \t\t\t" + d1.getTime());
        System.out.println("d1 suit d0bis: " + d1.after(d0bis));
        d1.add(Calendar.DAY_OF_MONTH, -30);
        System.out.println("d1 retardée de 30 jrs \t\t\t" + d1.getTime());
        d1.roll(Calendar.DAY_OF_MONTH, 30);
        System.out.println("d1 traduite de +30 jrs \t\t\t" + d1.getTime());

        long diff = (d1.getTimeInMillis() -
                     d0.getTimeInMillis())/CONST_DURATION_OF_DAY;
        System.out.println("Nombre de jours entre d1 et d0: "+diff);

        // calcul du numéro de la semaine
        int n = d0.getMinimalDaysInFirstWeek();
        System.out.println("Semaine n° " + (((7 - n) +
                     d0.get(Calendar.DAY_OF_YEAR) - 1)/7) + 1));
    }
}

Résultats:

Date d0          Mon Nov 24 10:40:48 CET 2014
Date d0bis        Mon Nov 24 10:40:48 CET 2014
Date d0bis_modifiée   Wed Dec 24 10:40:48 CET 2014
Date d0bis_rétablissement Mon Nov 24 10:40:48 CET 2014
le 24 du 11 ème mois de l'année 2014 soit le 328 ème jour de l'année 2014

Date d1          Sat Mar 14 18:05:03 CET 2015
d1 suit d0bis: true
d1 retardée de 30 jrs   Thu Feb 12 18:05:03 CET 2015
d1 traduite de +30 jrs   Sat Feb 14 18:05:03 CET 2015
Nombre de jours entre d1 et d0: 82
Semaine n° 48
```

Comparaison de 2 dates

2 façons de créer une instance de Calendar

Passage de d0bis au mois de Décembre

Rétablissement de la date initiale

Affichage détaillé de la date

Avancement calendrier de 30 jours, puis recul de 30 jours du seul champ DAY