Université de Nice - Sophia Antipolis
Faculté des Sciences

DEUG MIAS MP1

Programmation 2001-02

2. PROGRAMMEZ AVEC CLASSE

A. JAVA un langage de programmation objet.

Java est un langage de programmation objet. La programmation objet repose sur des langages objet qui permettent de représenter les objets de notre réalité (objets réels ex: une voiture, un étudiant, etc. ou objets conceptuels ex: une fonction mathématique, une liste de personnes, un compte en banque, etc.) par des objets informatiques (objets virtuels, représentés en machine par un identificateur unique, des attributs et leur valeur, des méthodes i.e. des procédures ou fonctions permettant d'interagir avec l'objet et de lui envoyer des messages). Ces objets informatiques permettent ensuite de représenter, manipuler ou simuler les objets réels à travers des représentations informatiques. Cette approche repose sur une dichotomie entre deux entités : les classes d'objets et les instances d'objets.

Définition : Une **classe** est une entité des langages objets qui représente une catégorie d'objets ayant des points communs (des variables d'instances, des méthodes, etc.). Cette classe a un nom et une description. La description donne les variables d'instance avec leur type ainsi que les méthodes d'instance que possèdent chacun des objets de cette classe. Une fois qu'une classe a été décrite, elle peut être utilisée comme un moule pour créer autant d'objets que voulu. Les classes permettent ainsi d'ajouter de nouveaux types complexes au langage. La description donne aussi les variables de classe et les méthodes de classe aussi appelées variables et méthodes statiques.

Définition : une **variable statique** est une variables liée à la classe elle-même et non à chaque objet. On y accède soit directement à travers la classe (si elle est en accès publique) soit à travers des accesseurs et des modificateurs statiques c'est à dire des méthodes statiques de la classe permettant respectivement de consulter et de modifier cette valeur.

Définition : une **variable d'instance** est une variables liée à un objet. On y accède soit directement à travers la l'instance de l'objet (si elle est en accès publique) soit à travers des accesseurs et des modificateurs c'est à dire des méthodes d'instances permettant respectivement de consulter et de modifier cette valeur.

Définition : une **méthode statique** est une méthode qui s'invoque à partir de la classe elle-même et non pas à partir d'un de ses objets.

Définition : une **méthode d'instance** est une méthode qui s'invoque à partir d'un objet et dont l'exécution se répercute sur cette instance.

Définition : Une **instance d'objet** est un objet individuel et identifié de manière unique par une référence en mémoire. L'objet possède les variables d'instance et les méthodes d'instance définies par sa classe. Les valeurs de ses variables d'instances lui sont propres et indépendantes des valeurs de celles des autres instances. L'appel d'une méthode d'instance sur un objet se répercute sur cet objet uniquement.

Exemple "les voitures": Il existe des classes de voitures : les coupés, les berlines, les monospaces, etc. Chaque classe représente des voitures ayant une particularité en commun par rapport à celles qui n'appartiennent pas à leur classe. Il existe évidemment des instances de ces classes: "la 306 blanche immatriculée 3195 SH 45", "la Renaud Espace 2879 XV 06", etc. Dans les langages objets on peut représenter cette réalité par des classes d'objets informatiques représentant les catégories de voitures et des objets instances de ces classes représentant des voitures existantes.

Définition : Le fait de créer un objet appartenant à une classe s'appelle **l'instanciation**: on **instancie** la classe et l'objet obtenu est une **instance** de la classe choisie. L'objet **instancié** possède toutes les caractéristiques décrites par sa classe.

Définition : Un **constructeur** est une méthode appelée lors de la création d'un objet à partir d'une classe. Le constructeur est une méthode décrite dans la classe et portant le nom de la classe.

Nous allons faire un petit exemple complet. Soit la classe des ordinateurs, chaque ordinateur a une marque de fabricant, que l'on veut pouvoir lire et modifier. On veut aussi pouvoir connaître le nombre d'ordinateurs ayant été créés. Voici une programmation d'un modèle objet de cette vision de la réalité du domaine des ordinateurs:

```
class Ordinateur {     // Déclaration de la classe, par défaut elle hérite de Object

  String marque;       // Déclaration d'une variable d'instance de type string

  static int nbOrdinateurs = 0;      // Déclaration d'une variable de classe de type int

  public Ordinateur(String laMarque) {
                     // Constructeur i.e. méthode appelée à la construction d'un objet
       this.setMarque(laMarque);     // initialiser la marque
       nbOrdinateurs++;              // compter les ordinateurs
  }

  public void setMarque (String laMarque) {  // Modificateur de la marque
       this.marque=laMarque;   // modifie la marque de l'instance appelant la méthode
  }

  public String getMarque () {               // Accesseur de la marque
       return(marque);
  }

  public static int getNbOrdinateurs() {    // Accesseur statique pour le nb d'ordinateurs
       return(nbOrdinateurs);
  }
}
```

ⓘ Le mot clef `class` permet de déclarer une classe, il est suivit de son nom et de la description de la classe entre accolades .

A l'intérieur des accolades on déclare les variables et les méthodes d'instances et/ou de classe. Par convention les noms des variables et des méthodes commencent toujours par une minuscule (à l'exception des constructeurs).

ⓘ Le mot clef `static` permet de déclarer une variable ou une méthode de classe (c'est à dire statique).

Ici la variable `nbOrdinateurs` et la méthode `getNbOrdinateurs` appartiennent à la classe et non à ses objets.

Un constructeur est souvent utilisé pour initialiser les variables d'une instance lors de sa création. Ici **`Ordinateur`**`(String laMarque)` indique que pour créer un objet de la classe `Ordinateur` il faut donner sa marque.

ⓘ Dans une méthode d'instance, le mot clef `this` représente l'objet courant c'est à dire l'objet à travers lequel la méthode a été appelée. Le mot clef `this` permet ainsi d'accéder aux méthodes et aux variables d'instances de l'objet à travers lequel est invoquée la méthode. L'instruction `this(...)` permet d'appeler les constructeurs de l'objet courant.

L'instruction **`this`**`.setMarque(laMarque);` appelle donc la méthode d'instance `setMarque` sur l'objet créé.

ⓘ Un modificateur est une méthode permettant de modifier une variable. Un accesseur est une méthode permettant de consulter une variable. On préfère cette façon d'accéder aux variables afin de préserver l'intégrité des objets.

La méthode `setMarque (String laMarque)` est un modificateur : c'est une méthode permettant de modifier la valeur de la variable d'instance `marque`. La méthode `String getMarque()` est un accesseur : c'est une méthode permettant de consulter la valeur de la une variable d'instance `marque`.

ⓘ Le mot clef `public` indique que l'accès à la méthode ou à la variable est illimité, nous ne verrons pas cette année les autres restrictions d'accès (`private`, `protected`).

Exercice 2.1 Tester l'exemple des ordinateurs

a) Implantez cette classe, compilez et corrigez vos erreurs
b) Créez une variable `mac` et utilisez l'instruction `new` pour l'initialiser avec un objet de la classe Ordinateur. Le constructeur attend un paramètre qui sera par exemple `"iMac Platinium"`.
c) Consultez la marque de votre objet `mac` en utilisant son accesseur (méthode d'instance).
d) Consultez le nombre d'ordinateurs en utilisant son accesseur.
d) Créez une nouvelle variable `pc`, initialisez-la puis consultez la marque des deux objets et le nombre d'ordinateurs.

Exemple d'utilisation:

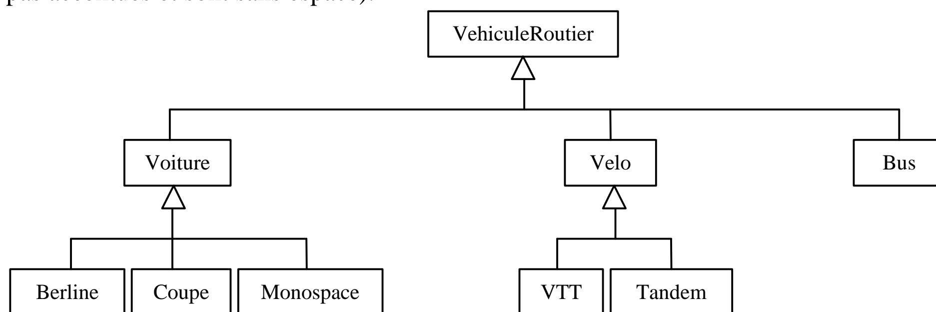
```
Welcome to DrJava.
> Ordinateur mac=new Ordinateur("iMac Platinium");
> mac.getMarque()
iMac Platinium
> Ordinateur.getNbOrdinateurs()
1
> Ordinateur pc=new Ordinateur("Compaq Aramada E550");
> pc.getMarque()
Compaq Aramada E550
> mac.getMarque()
iMac Platinium
> Ordinateur.getNbOrdinateurs()
2
```

ⓘ Le mot clef `new` permet donc de créer un nouvel objet (`new` permet l'instanciation)

Lors de sa création le constructeur spécifié après `new` est appelé avec les paramètres donnés. La référence à l'objet créé est renvoyée par `new` et peut être stokée dans une variable, par exemple ici : `mac` et `pc`.

Les classes peuvent être organisées selon une hiérarchie. La notion est très proche de la taxonomie en biologie: depuis Aristote, il y a 2300 ans, on utilise les taxinomies (catégorisation hiérarchique basée sur les similarités entre objets) qui se représentent par un arbre de catégories partant d'une catégorie générale (ex: il existe la catégorie des êtres vivants) et se raffinant progressivement (ex: la catégorie des êtres vivants inclut deux sous catégories, les animaux et les végétaux) et récursivement pour arriver à des catégories très précises (ex: la catégorie des animaux inclut plusieurs sous catégories, les mammifères, les reptiles, les insectes, etc. ; la catégorie des reptiles inclut celle des crocodiles, etc.)

De la même façon, les classes d'objets informatiques peuvent être organisées selon une hiérarchie de spécialisation : toutes les sous classes d'une classe héritent des caractéristiques de la classe supérieure. Par conséquent, tous les objets d'une sous classe hériteront des caractéristiques de leur classe et de celles qui sont au-dessus dans le hiérarchie. Pour exemplifier nous allons durant tout le TP travailler avec la hiérarchie suivante (Notez que les noms de classes ne sont pas accentués et sont sans espace):



Ainsi si l'on crée un objet `Laguna2357SH45` comme une instance de la classe `Berline`, il héritera de toutes les caractéristiques de la classe `Berline` mais aussi de celles de la classe `Voiture` ou `VehiculeRoutier`.

Définition : une classe définie une catégorie d'objet. On dit qu'**une classe A étend une classe B** lorsqu'elle définit une sous catégorie de B. Tous les objets de A sont alors aussi des objets de B.

Définition : on appelle **héritage** le mécanisme par lequel un classe possède au moins toutes les variables et toutes les méthodes de la classe dont elle descend.

Définition : on appelle **surcharge** ou **redéfinition** le fait de redéfinir (réécrire) une méthode héritée d'une classe ancêtre (classe dont on hérite).

B. Mettre de l'ordre dans ses classes

Nous allons implanter en Java la hiérarchie de classes donnée précédemment et munir ces classes de méthodes et de variables pour chacun de leurs objets. Le code suivant est un exemple de déclaration de classes représentant la branche des vélos:

```
class VehiculeRoutier {

}

class Velo extends VehiculeRoutier {

}

class VTT extends Velo {

}

class Tandem extends Velo {

}
```

ⓘLe mot clef `extends` indique que la nouvelle classe que l'on est en train de déclarer, étend une classe existante, c'est à dire que cette nouvelle classe est plus précise, i.e. qu'elle est en-dessous dans la hiérarchie.

Une classe étendant une autre classe héritera donc des caractéristiques de la classe dont elle descend. Si le mot clef `extends` est absent la classe n'a pas explicitement de classe supérieure. Cependant dans ce dernier cas, Java la place en-dessous de la classe la plus générale qui existe: la classe `Object`. Donc en pratique toutes les classes descendent au moins de la classe `Object` et héritent par conséquent de toutes ses caractéristiques.

La classe `Object` définit entre autres une méthode `public String toString()` qui permet de calculer une chaîne de caractères représentant un objet. Elle est utilisée en particulier lorsque l'on veut afficher un objet. Cette méthode est donc héritée par nos objets et c'est elle qui est appelée lorsque l'on les affiche. La méthode de la classe `Object` se contente d'afficher le type et la référence en mémoire.

De plus si la classe ne définit aucun constructeur, elle hérite du constructeur vide de la classe supérieure. Ici nos classes vont donc hériter du constructeur sans paramètre de la classe Object. Par conséquent on peut créer des objets très simples appartenant à ces classes en utilisant un constructeur sans paramètres.

Exercice 2.2 Implanter la hiérarchie des classes
a) Créez un fichier par classe ayant pour nom le nom de la classe suivi de **.java**. Sauvegardez les fichiers dans un répertoire E:\Etudiant\Vehicules\. Vous n'oublierez pas de les recopier sur votre disquette avant de partir ☺
b) Recopiez l'exemple précédent, **compilez**, corrigez vos erreurs si nécessaire.
c) Dans le top-level, créez un objet de la classe `VTT` et affichez le (c'est moche hein!)
d) Complétez l'exemple pour implanter la hiérarchie complète des véhicules, **recompilez**, corrigez vos erreurs si nécessaire.
e) Dans le top-level, créez un objet de la classe `Berline`, et un de la classe `Bus` et affichez les. Expliquez l'affichage ? avez-vous compris le mécanisme permettant d'obtenir cet affichage ?

C. La richesse de l'héritage

Les classes que nous venons de créer sont vides. Pour l'instant elles ne permettent rien d'autre que de typer les objets que l'on crée en disant 'tel objet est une instance de telle classe'. En vous souvenant du premier semestre (hou la la !) vous allez compléter la classe `VehiculeRoutier`. Ce faisant les autres classes vont hériter de ce que vous ajoutez à l'exception des constructeurs qui doivent être programmés pour chaque classe.

Exercice 2.3 Compléter la classe VehiculeRoutier
a) Ajoutez des variables d'instances à la classe `VehiculeRoutier` pour mémoriser le nom, l'année d'achat et la capacité maximale en passagers de chacun des véhicules. (`nom`, `anneeAchat`, `capacitePassagers`)
b) Ajoutez les accesseurs à la classe `VehiculeRoutier` permettant de consulter chacune des ces valeurs (des fonctions `getNom()`, `getAnneeAchat()`, `getCapacitePassagers()`) ainsi qu'un modificateur permettant de modifier la capacité en passagers (une procédure `setCapacitePassagers(...)`).
c) Compilez, corrigez vos erreurs si nécessaire. Pour tester, dans le top-level, créez un objet de la classe `Voiture`. En utilisant les accesseurs et les modificateurs: affichez sa capacité, modifiez sa capacité à 5 et affichez la à nouveau.

Vous l'aurez remarqué, les méthodes que nous avons créées ne permettent pas de modifier le nom et l'année d'achat. Ces valeurs seront en fait données une fois pour toute lors de la création d'un véhicule. Nous avons dit que le constructeur est la méthode appelée lors de la création d'un objet. Les constructeurs portent le nom de leur classe et peuvent varier de par les arguments qu'ils attendent. Les constructeurs ne s'héritent pas, il faut donc les créer pour chaque classe. La seule exception est le constructeur vide qui est hérité par défaut si aucun autre constructeur n'est défini. Dès qu'un constructeur est défini pour une classe celle-ci n'hérite plus du constructeur vide.

ⓘ Le mot clef `super` permet d'accéder aux méthodes de la classe supérieure (celle dont on hérite). Par exemple `super.toString();` appelle la méthode `toString()` définie dans la classe supérieure. En l'utilisant directement comme une méthode `super(...)` permet d'accéder aux constructeurs de la classe supérieure.

Exercice 2.4 Créer les constructeurs

a) Ajoutez un constructeur à la classe `VehiculeRoutier` qui prend en arguments le nom et l'année d'achat et qui initialise les variables d'instance `nom` et `anneeAchat` avec ces valeurs. Pour cet exercice nous ignorerons l'initialisation de la capacité en passagers. Comme c'est un entier elle est, par défaut, à la valeur 0.
b) Ajoutez un constructeur aux autres classes, prenant les mêmes arguments pour en faire la même chose, mais programmez le intelligemment en utilisant `super(...)`. Rappelez-vous que par héritage, toutes les classes de véhicules ont maintenant des variables, des accesseurs, et un modificateur.
c) Enregistrez et compilez **toutes** les classes, corrigez vos erreurs si nécessaire.
d) Pour tester, dans le top-level, créez un objet de la classe `Velo` et un objet de la classe `Bus`, consultez les valeurs de leurs variables d'instance `nom` et `anneeAchat` grâce à leur accesseur. Enfin affichez les objets que vous avez créés et remarquez que l'affichage est toujours aussi laid.

L'affichage de nos objets n'est toujours pas terrible. Nous avons vu qu'en Java, toutes les classes descendent par défaut d'une classe unique, la classe `Object`. Par conséquent elles héritent toutes de cette classe.

La classe `Object` définit la méthode `public String toString()`qui est héritée par nos objets et c'est elle qui est appelée lorsqu'on les affiche. Cette méthode de la classe `Object` se contente d'afficher le type et la référence en mémoire. Pour améliorer cet affichage nous allons surcharger (redéfinir) cette méthode pour qu'elle renvoie quelque chose de plus présentable. Si cette méthode est qualifiée de `public` c'est justement pour que l'on puisse y accéder de l'extérieur et la redéfinir. Par défaut, lorsque Java appelle une méthode sur d'un objet, il utilise la définition la plus précise. Donc si nous redéfinissons cette méthode dans l'une de nos classe, cette définition étant plus précise, elle sera invoquée lors de l'affichage de l'objet.

Exercice 2.5 Surcharge de toString()

a) Ajoutez une nouvelle méthode `public String toString()` à la classe `VehiculeRoutier`. Cette méthode va écraser celle héritée de `Object`. Elle devra renvoyer une chaîne de caractères donnant le nom, l'année d'achat, la capacité et le type par exemple:

```
'Mazda mx5' année: 2001 capacité passagers: 0 type : Véhicule Routier
```

b) Ajoutez une nouvelle méthode public `String toString()` aux autres classes ; cette méthode récupère le résultat de la méthode de la classe supérieure, ajoute le nom de sa classe au bout (ex: " `Coupé`") et renvoie le résultat. *Quand vous lisez "la méthode récupère le résultat de la méthode de la classe supérieure" pensez au mot clef* `super`.
c) Enregistrez et compilez **toutes** les classes, corrigez vos erreurs si nécessaire.
d) Pour tester, dans le top-level, créez un objet de la classe `Coupe` et affichez le. Vous devriez obtenir quelque chose comme:

```
'Peugeot 207cc' année: 2002 capacité passagers: 0 type : Véhicule Routier Voiture Coupé
```

e) Expliquez le mécanisme qui amène à ce résultat

Nous avons vu que les sous classes héritent des méthodes et des variables de leur classe mère et ce récursivement. Les objets d'une classe sont donc compatibles avec les classes supérieures à la leur. Par conséquent ce sont aussi des objets des classes supérieures ex: un objet de la classe `VTT` est aussi un objet de la classe `Velo`, un objet de la classe `VehiculeRoutier` et un objet de la classe `Object`. Ceci nous permet de définir des méthodes générales manipulant les objets d'une classe ou de ses sous classes: une méthode définie en utilisant un certain type/classe est aussi applicable aux sous types/classes de celui-ci.

Exercice 2.6 Héritage de méthodes et propagation des types
a) Ajoutez une nouvelle méthode public `boolean plusJeuneQue(VehiculeRoutier unAutreVehicule)` à la classe `VehiculeRoutier`. Cette méthode permet de comparer l'année du véhicule à travers lequel la méthode est invoquée à celle du véhicule passé en paramètre. Si le véhicule passé en paramètre est plus vieux la méthode renvoie `true` sinon elle renvoie `false`.
b) Est-il nécessaire de définir cette méthode dans les autres classes ?
c) Compilez, corrigez vos erreurs si nécessaire.
d) Dans le top-level, créez un objet de la classe `Berline` et un objet de la classe `Tandem` ayant des années d'achat différentes. Pouvez-vous comparer leur année d'achat ? Comment faut-il faire ?

Plus nous raffinons les classes, i.e. plus nous descendons dans la hiérarchie, et plus il est possible de donner des valeurs ou des comportements par défaut aux objets. C'est le cas pour la capacité maximale en passagers que nous avons jusque là délaissée: certaines classes sont suffisamment précises pour que cette capacité maximale soit constante pour leurs objets alors que pour d'autres classes cette capacité maximale reste variable d'une instance à l'autre:

Classe de véhicules	Capacité maximale
`Berline`	Constante = 5
`Coupe`	Constante = 4
`Monospace`	Constante = 8
`Velo`	Constante = 1
`VTT`	Constante = 1
`Tandem`	Constante = 2
`Voiture`	Variable
`Bus`	Variable
`VehiculeRoutier`	Variable

Exercice 2.7 Valeurs par défaut et désactivation par surcharge
a) Pour les classes dont la capacité maximale est variable, ajoutez un constructeur afin qu'il admette un paramètre supplémentaire permettant d'initialiser la capacité. (n'oubliez pas que vous pouvez utiliser `this(...)` et `super(...)`)
b) Pour les classes dont la capacité est constante modifiez le constructeur afin qu'il mette à jour cette capacité en utilisant la valeur par défaut donnée dans le tableau ci-dessus (utilisez `super(...)`). De plus afin que l'on ne puisse pas modifier cette capacité, utilisez le mécanisme de la surcharge pour désactiver (pour ces classes uniquement) le modificateur pour la capacité et affichez le message "`Accès refusé: capacité constante`".
Attention : réfléchissez bien pour `VTT` et surtout pour `Tandem`...
c) Enregistrez et compilez **toutes** les classes, corrigez vos erreurs si nécessaire.
d) Dans le top-level, créez un objet de la classe `Tandem` et un objet de la classe `Bus`. Affichez les. Appelez la méthode de modification de la capacité pour le `Tandem` et pour le `Bus`. Affichez les à nouveau pour vérifier le résultat.

D. Méthodes et champs de classe

Nous avons vu que les classes peuvent avoir des méthodes et des variables qui leur sont propres (statiques), c'est à dire ces méthodes et ces variables ne participent pas à la description d'un moule pour créer des objets mais qui sont attachées à la classe et ne sont accessibles qu'à travers elle. Le mot clef `static` précédant leur déclaration, permet de les dissocier des méthodes et variables d'instance. Une variable ou une méthode statique est héritée par les sous classes et peut être surchargée.

Exercice 2.8 Méthodes et champs statiques
a) En vous inspirant de l'exemple donné en section A sur les ordinateurs, mettez en place un moyen pour compter le nombre de véhicules créés quelque soit leur type. (un variable statique `nbVehiculesRoutiers`, un accesseur statique `getNbVehiculesRoutiers` et une modification du constructeur).
b) Compilez, corrigez vos erreurs si nécessaire.
c) Dans le top-level, créez plusieurs véhicules de différents types et demandez le nombre de véhicules créés.

Une autre utilisation très courante des variables statiques, est la définition de constantes.

ⓘ Une constante est un champ dont la déclaration commence par les mots clefs `final static` qui signifient qu'il ne peut être modifié et qu'il est attaché à la classe et non aux instances.

Nous allons donc utiliser cela pour enrichir nos classes avec une variable d'instance notant si l'énergie du véhiculer est "`musculaire`", "`diesel`", "`essence`", "`G.P.L.`" ou "`électrique`". Ces constantes seront définies dans la classe `VehiculeRoutier`. Les constantes on des noms en majuscules, ici: `MUSCULAIRE`, `DIESEL`, `ESSENCE`, `GPL` et `ELECTRIQUE`. Pour la majorité des véhicules l'énergie est variable d'une instance à l'autre, mais certaines classes sont suffisamment fines pour que celle-ci soit constante:

Classe de véhicules	Energie
`Berline`	Variable
`Coupe`	Variable
`Monospace`	Variable
`Velo`	energie = "`musculaire`"
`VTT`	energie = "`musculaire`"
`Tandem`	energie = "`musculaire`"
`Voiture`	Variable
`Bus`	Variable
`VehiculeRoutier`	Variable

Exercice 2.9 Méthodes et champs statiques
a) Ajoutez une variable `energie` de type `String` à la classe `VehiculeRoutier` pour mémoriser l'énergie utilisée. Ajoutez aussi un accesseur et un modificateur pour ce nouveau champ.
b) Définissez des constantes contenant les valeurs possibles pour l'énergie.
c) Utilisez ces constantes pour vérifier la valeur du paramètre du modificateur et le cas échéant afficher le message "`Modification refusée: énergie inconnue.`". On rappelle que la comparaison entre chaînes de caractères se fait par la méthode `chaine1.equals(chaine2)`.
d) Pour les classes dont l'énergie est variable ajoutez un constructeur permettant de spécifier l'énergie. Pensez à `super(...)` et à `this(...)` et utilisez l'accesseur afin de bénéficier de la vérification qu'il fait avant d'autoriser le changement de l'énergie d'un véhicules.
e) Pour classes ayant une énergie unique, quelle modification simple (et déjà utilisée précédemment) faut-il apporter pour assurer une valeur par défaut constante pour les objets de ces classes ? Est-il nécessaire de modifier plusieurs classes ?
f) Dans le top-level, créez plusieurs véhicules de différents types. Utilisez les constantes (ex: `VehiculeRoutier.DIESEL`) pour modifier leur énergie. Vérifiez le comportement des instances des classes à énergie unique.

E. Ensembles d'objet et polymorphisme

Pour manipuler des ensembles d'objets, soit on crée une variable par objet (dans le meilleur des cas cela peut être fastidieux et le plus souvent c'est impossible par exemple lorsque l'on ne sait pas à l'avance combien d'objets seront créés) soit on utilise des objets particuliers (listes, ensembles, tableaux etc.) qui permettent d'emmagasiner et de manipuler un nombre arbitraire d'objets. Parmi les classes prédéfinies en Java pour la manipulation d'un ensemble d'objets, nous allons en utiliser deux: les `HashSet` (qui permettent de manipuler un ensemble d'objets) et les `Iterator` (qui permettent de parcourir un ensemble d'objets).

La classe `java.util.HashSet` permet de créer un ensemble d'objet. Il possède un constructeur sans arguments et propose plusieurs méthodes dont deux nous intéressent:

- `boolean add(Object o)` qui permet d'ajouter un objet à l'ensemble
- `Iterator iterator()` qui permet d'obtenir un itérateur sur l'ensemble

La classe `java.util.Iterator` permet de créer un objet permettant de parcourir chacun des éléments d'un ensemble (comme la tête de lecture d'un CD est un objet qui parcourt les chansons sur votre CD). Cette classe propose deux méthodes qui nous intéressent :

- `boolean hasNext()` qui permet de savoir s'il reste encore des éléments à parcourir
- `Object next()` qui permet d'obtenir un nouvel objet pas encore visité

Nous allons utiliser ces classes pour définir un objet représentant une flotte de véhicules et des méthodes pour gérer cette flotte.

Exercice 2.10 Ensemble d'objets

a) Ajoutez la classe suivante au fichier, compilez pour vérifier.

```
class FlotteVehicules extends HashSet {

  public String toString() {
    String leResultat = new String();
    Iterator iter=this.iterator();
    while(iter.hasNext())
       leResultat = leResultat +" ["+iter.next().toString()+"]\n";
    return(leResultat);
  }

}
```

b) Commentez chaque ligne pour expliquer ce que l'on y fait.
Comment `HasSet` est-il utilisé? Pourquoi ?
Regardez bien l'emploi de l'`Iterator` car nous allons le réutiliser plusieurs fois.

c) Dans le top-level créez une flotte de véhicules contenant 6 véhicules différents et affichez la.

ⓘ L'opérateur `instanceof` est un opérateur binaire booléen qui vérifie si l'objet à sa gauche est compatible avec la classe à sa droite.

Exercice 2.11 Vérification de la classe d'un objet:

Créez un objet de la classe `Velo` et vérifiez s'il appartient aux classes, `Velo`, `Bus`, `VTT` et `VehiculeRoutier`

La méthode `Object next()` de l'`Iterator` renvoie un objet de la classe `Object`. Ceci est ennuyeux si l'on veut appeler des méthodes de la classe `VehiculeRoutier` qui ne sont pas définies dans `Object`. Cependant si l'on connaît la classe d'un objet (ou une classe supérieure) on peut utiliser l'opérateur de forçage de type pour raffiner son type.

ⓘ L'opérateur de forçage de type ou opérateur de cast s'écrit `(NOUVEAU TYPE)` et se place devant l'expression à convertir. Une erreur se produit si la conversion n'est pas possible par exemple si on essaie de changer la classe d'un objet pour une classe qui ne fait pas partie de l'héritage de l'objet

Exercice 2.12 Cast et vérification du type

a) Ajoutez à la classe `FlotteVehicules` la fonction `public String capacitePour(int p_NbPassagers)` qui renvoie les descriptions des véhicules capables de transporter le nombre de passagers donné en paramètre. Inspirez-vous de la méthode `public String toString()` donnée précédemment et utilisez l'opérateur de cast là où cela est nécessaire

b) Compilez et corrigez vos erreurs si nécessaire.

c) Dans le top-level créez une flotte de véhicules contenant 6 véhicules différents et affichez successivement ceux qui sont capables de transporter 1, 3, 7 et 12 personnes.

d) Est-il possible d'ajouter un objet de la classe `String` à la flotte ? essayez ? cela pose-t-il un problème ?

e) Pour empêcher cet ajout, que faut-il faire ? quelle méthode modifier et comment ?

PS: Il vous reste du temps ? Programmez la méthode `public String chercher(String p_MotClef)` permettant de chercher dans la liste des véhicules ceux dont la description contient le mot clef donné en paramètre. Exemple:

```
> MaFlotte.chercher("Renaud")
```