Exercice 1

Nous désirons développer un programme pour la gestion (très simplifiée) d'un parc de véhicules destinés à la location.

Pour chaque véhicule, il est demandé de renseigner le *modèle*, *l'année d'achat*[1], le *prix d'achat* capable de gérer les centimes, le *numéro d'immatriculation* et le *permis* (une lettre) nécessaire à la conduite de ce véhicule.

Plusieurs types de véhicules sont possibles à la location, certains ayant des caractéristiques particulières. Ainsi, il est demandé de représenter l'information comme quoi une voiture dispose ou non d'un *autoradio*. Pour la location d'un camion, le *volume* de stockage possible du camion doit être précisé. Enfin, des autocars peuvent aussi être loués, et pour ces véhicules il est nécessaire de connaître leur *volume* (pas forcément entier) de stockage possibles ainsi que le *nombre de passagers* qu'ils peuvent transporter.

Modélisation

1. Dessiner l'arbre d'héritage de classes, où figurera les noms des classes, les attributs et leurs types, et les liens d'héritage.
2. Traduire la définition des classes en Java. Prévoyez un constructeur pertinent pour chaque classe.
3. Instancier dans un programme principal (méthode "`main`" d'une classe, la classe *Test* par exemple) chacune de ces classes.

Codage des méthodes

1. Définir un constructeur qui initialise les différents champs de chacune des classes.
2. Écrire les méthodes `ajouterAutoradio()` et `enleverAutoradio()` qui permettent de fixer la valeur de l'attribut autoradio d'une voiture.
3. Écrire la méthode `age()` qui retourne l'âge d'un véhicule (en nombre d'années). L'année actuelle sera écrite en dur dans le code.
4. Écrire la méthode `infoVehicule()` qui retourne une chaîne de caractères (**`String`**) caractérisant un véhicule. Écrire la méthode `infoVoiture()` qui retourne une chaîne de caractères caractérisant une voiture.
5. Écrire une méthode `peutTransporterVolume(…)` qui, à partir d'un volume donné en paramètre, retourne vrai ou faux selon que le camion peut ou non permettre le transport de ce volume.
6. Écrire une méthode `coutLocation(…)` qui calcule le coût quotidien de location d'un véhicule. Ce coût est calculé comme suit : pour les véhicules de moins d'un an, le coût de location est le prix d'achat du véhicule / 200, et pour les véhicules plus anciens, le coût de location est le prix d'achat / 250.
7. Écrire une méthode `peutTransporterPassagers(…)` dans la classe adéquate qui, à partir d'un nombre de passagers et d'un volume moyen par passager (représentant les bagages), retourne vrai ou faux selon que l'autocar peut ou non transporter ces passagers.
8. Écrire la méthode `infoCamion()` qui retourne une chaîne de caractères caractérisant un camion.
9. Écrire la méthode `infoAutocar()` qui retourne une chaîne de caractères caractérisant un autocar.

Classe contenant le programme principal

1. Écrire/compléter un programme de test créant une (instance de la classe) voiture Twingo, achetée cette année 10000 €, immatriculée 1234 AZ 49, avec un autoradio et nécessitant un permis B. Puis afficher sur la sortie standard (en utilisant la méthode **`System.`***`out`*`.println(…)`) la chaîne de caractères retournée par la méthode `infoVoiture()`. Le résultat est-il celui escompté ?

[1] La classes `Date` ou `Calendar` existent en Java, mais le but du TP n'est pas de les utiliser. Nous représenterons une date simplement par un entier représentant son année.

2. Créer un camion J9, acheté 20000€ il y a 5 ans, immatriculé 987 BCD 75, utilisable avec un permis B, et pouvant transporter 25 m³. Afficher sur la sortie standard la chaîne de caractères retournée par la méthode `infoCamion()`. Afficher si ce camion peut transporter 7 m³, on utilisera pour cela un appel à la méthode `peutTransporterVolume(…)`.
3. Afficher le coût de location de la Twingo précédemment créée, ainsi que celui du camion de type J9.
4. Créer un autocar de type FRI, acheté 90000€ en 2005, immatriculé 4567WX01, nécessitant un permis D, permettant de transporter 53 passagers et disposant d'une soute à bagages de 3 m³. Affichez si cet autocar peut transporter 40 passagers ayant chacun 0,1 m³ de bagages.
5. Les classes de l'application sont appelées à être complétées et utilisées par plusieurs programmeurs. Quelle solution proposez-vous, en ce qui concerne les champs (attributs et méthodes), pour assurer cette gestion ? Implémentez-là.

Exercice 2

1. Créer les classes `Agence`, `Bureau` et `Garage` permettant la représentation des bureaux de location et garages **ainsi que les méthodes demandées en TD**. On utilisera un tableau de références sur des garages pour représenter l'ensemble des garages connus par un bureau de location. Un bureau pouvant connaître 4 garages ou moins, nous utiliserons un tableau de 4 références sur des garages en utilisant la valeur de référence particulière **`null`** pour marquer les « cases vides » du tableau.
2. Écrire une (des) méthode(s) `afficherXXX()` produisant des affichages tels que :

```
Bureau de location - Angers - 0241123456 - 4 adm - 5 comm
Garage - Angers sud - 0241234567 - 1 adm - 6 meca
```

Exercice 3

1. Ajouter aux classes représentant les véhicules et les agences des méthodes `toString()` retournant l'équivalent de ce qui est affiché par `afficherXXX()` pour les agences et renvoyant la même chose que les méthodes `infoXXX()` pour les véhicules.
2. Modifier en conséquence les méthodes `afficherXXX()` en utilisant les méthodes `toString()`, et vérifier qu'il est possible d'utiliser des objets `Vehicule` ou `Agence` comme paramètre de **`System.`***`out`*`.println()`.
3. Écrire un programme de test qui crée un bureau de location et trois garages, ajoute ces garages au bureau de location, et affiche les différents objets créés par un **`System.`***`out`*`.println()`.
4. Écrire une méthode `categorie()` qui retourne un entier représentant la catégorie de l'agence. Les agences de catégorie 1 sont celles employant moins de 10 personnes, les agences de catégorie 2 sont celles en employant 10 à 19, et les agences de catégorie 3 sont celles en employant 20 ou plus. Cette méthode ne devra être définie que dans une seule classe.
5. Écrire une méthode `compareTo()` prenant une agence en paramètre qui compare la taille de cette agence avec l'agence courante : cette méthode devra retourner (une référence sur) celle qui emploie le plus de personnel.
6. Écrire une méthode `afficherGarages()` (dans la classe représentant les bureaux) prenant comme paramètre un entier `cat` et affichant les garages, liés à ce bureau, dont la catégorie est supérieure ou égale à `cat`. L'affichage des caractéristiques des garages devra être précédé d'une ligne décrivant ce qui est affiché, comme sur l'exemple suivant :

```
Garages de catégorie >= 2 du Bureau de location - Angers - 0241123456
- 4 adm - 5 comm
Garage - Angers est - 0241456789 - 5 adm - 10 meca
```

Annexes

Types primitifs

Un entier se note **`int`**, un entier long **`long`**, un réel **`float`** ou **`double`** et un booléen se note **`boolean`** ayant pour valeur **`true`** ou **`false`**. Une chaîne de caractères est représentée par une instance de la classe **`String`**... Il ne s'agit donc pas d'un type primitif.

Commentaires

```
// ceci est un commentaire sur une seule ligne
/* cela est
   un commentaire sur
   plusieurs lignes*/
```

Classes et attributs

Une classe se définit comme suit :

```
class NomClasse {
     // Attributs
     TypeAttribut nomAttribut;
     …
     // Méthodes de la classe
     …
}
```

Pour définir l'héritage d'une classe parent C par une classe SC, on définit SC de la façon suivante :

```
class SC extends C {
     // définition de la classe…
}
```

Une classe dont le nom est `NomClasse` doit obligatoirement être saisie dans un fichier de nom `NomClasse.java`. Pour compiler un tel fichier source Java, appeler le compilateur de la façon suivante :

```
javac NomClasse.java
```

Une fois la compilation exécutée, s'il n'y a pas d'erreur (auquel cas, les corriger), le compilateur a créé un fichier `NomClasse.class` qui contient la version compilée de la classe.

Méthodes

Les méthodes sont déclarées et définies à l'intérieur de la classe. La syntaxe de Java est très largement inspirée de celle du C :

- le type de retour d'une méthode est déclaré juste avant son nom ;
- pour définir une méthode qui ne retourne pas de résultat, le type de retour est **`void`** ;
- les paramètres sont signalés par leur type puis leur nom, et séparés d'une virgule ;
- l'affectation se fait par = et le test d'égalité par ==, les syntaxes de **`if`**, **`switch`**, **`while`** et **`for`** sont identiques.

```
class Exemple {
     void methode1() {
          System.out.println(”essai”);
     }
     int somme(int a, int b) {
          return a + b;
     }
     boolean positifP(int a) {
          if (a > 0)
               return true;
          else
               return false;
     }
}
```

Instanciation

Pour instancier une classe, c'est à dire créer un objet de cette classe, on doit déclarer une référence puis créer l'instance en utilisant **`new`** suivi d'un constructeur de la classe à instancier. Notez que la déclaration et la création de l'objet peut se faire sur une seule ligne.

```
NomClasse o;         // déclaration[2]
o = new NomClasse(); // création de l'objet
AutreClasse o2 = new AutreClasse(...);
```

Il est évidemment possible d'utiliser une référence comme type d'un attribut d'une classe, comme type d'un paramètre de méthode ou comme type de retour d'une méthode.

[2] Quand une référence est déclarée, elle est (implicitement) initialisée à la valeur `null`.

```
class Exemple {
     AutreClasse objetMemorise;
     AutreClasse obtenirValeur() {
          return objetMemorise;
     }
     void fixerValeur(AutreClasse i) {
          objetMemorise = i;
     }
}
```

Référence `null`

`null` est une valeur de référence particulière (non typée, pouvant donc être utilisée avec tout type de référence) qui ne repère aucun objet. Quand une référence est créée, elle est implicitement initialisée à **`null`**, sauf indication contraire. Il est évidemment interdit d'accéder à des attributs ou d'appeler des méthodes à partir d'une référence **`null`**. Cela générera un avertissement à la compilation.

Constructeurs et appel du constructeur de la super-classe

Lorsqu'on définit une classe `C`, on peut définir des constructeurs avec ou sans paramètres pour initialiser les objets créés. Tout constructeur d'une classe autre que celui de la classe **`Object`** (voir plus loin) fait appel soit à un constructeur de la super-classe, soit à un constructeur de la même classe. Cet appel doit être la première instruction dans la définition du constructeur. On utilise la syntaxe **`super`**`(arguments)` (où `arguments` peut être vide ou non) pour appeler un constructeur de la superclasse, et **`this`**`(arguments)` pour appeler un constructeur de la même classe. Attention, ces intructions doivent être les premières du constructeur.

Si aucun constructeur de la super-classe ou de la même classe n'est appelé explicitement, le compilateur génère implicitement un appel au constructeur par défaut **`super`**`()` qui est sans paramètre (il faut qu'il y ait un tel constructeur dans la super-classe sinon une erreur est détectée).

```
class Sc extends C {
     public Sc(String s) {
          // appel implicite au constructeur C()
          … // initialisation propre à Sc (gestion de s par exemple)
     }
     public Sc(int a, String s){
          super(a); // appel du constructeur C(int)
          … // initialisation propre à Sc (gestion de s par exemple)
     }
}
```

Appel de la super méthode

Quand une méthode est redéfinie dans une sous-classe, il est souvent nécessaire d'appeler la méthode masquée (méthode de même signature de la super-classe). Cet appel peut être fait n'importe où dans le code de le méthode de la sous-classe. L'appel se fait en utilisant la pseudo-variable **`super`** qui, à la manière de **`this`**, repère l'objet sur lequel la méthode est en train d'être exécutée, mais qui est du type de la super-classe. Ainsi, elle n'est pas nécessairement la première instruction de la méthode.

```
class Sc extends C {
     public void methode(int a) { /* redéfinition d'une méthode
                                     définie dans C */
          …
          super.methode(a);
          …
     }
}
```

La classe `Object` : racine de l'arbre d'héritage

Si une classe est déclarée comme n'ayant aucune super-classe, elle admet implicitement **`Object`** comme super-classe. La classe **`Object`** fait partie du paquetage `java.lang`, qui contient les classes et les interfaces les plus centrales du langage.

Getters / Setters

Afin de rendre les classes plus robustes, on peut interdire la modification libre pour vérifier que les champs correspondent bien à certains critères. Pour cela, on déclare les champs privés grâce à l'attribut **`private`** qui le rend inaccessible en dehors de la classe. Pour pouvoir accéder à ou modifier sa valeur, on utilise deux méthodes pour chacun des champs : un getter qui permet d'accéder à sa valeur et un setter qui permet de modifier sa valeur.

```
class C {
     private int champ1;
     private String champ2;

     public int getChamp1() {
          return this.champ1;
     }

     public void setChamp1(int champ1) {
          this.champ1 = champ1;
     }
}
```

Programme principal

Pour écrire un programme de test, il est nécessaire d'écrire une méthode particulière dans une nouvelle classe ou une des classes du projet. Cette méthode devra avoir la signature suivante :

```
class NomClasse {
     public static void main(String args[]) {
          …
     }
}
```

Une classe possédant une telle méthode `main` peut être « exécutée ». C'est à dire qu'il est possible de lancer l'exécution de cette méthode `main` en appelant la machine virtuelle Java de la façon suivante (dans le répertoire contenant le fichier `NomClasse.class` obtenu après compilation) :

```
java NomClasse
```

Tableaux

En Java, on ne manipule pas directement des tableaux (comme en C ou en Pascal), mais des références sur des tableaux. Ainsi la déclaration **`int`**`[] tab;` (la syntaxe **`int`** `tab[];` est aussi acceptée) déclare une référence sur un tableau d'entiers. Pour pouvoir utiliser un tableau, il faut donc le créer (de la même façon qu'il faut instancier une classe pour utiliser un objet, et non pas simplement déclarer une référence sur un objet) en précisant sa taille : `tab =` **`new int`**`[20];`. Ceci crée un tableau de 20 entiers, ces entiers étant accessibles par `tab[0]` ... `tab[19]`. Contrairement au langage C, il est possible de connaître la taille d'un tableau, en accédant à l'attribut `length` d'un tableau (**`System`**`.`*`out`*`.println(tab.length);` affichera 20). Attention : cet attribut est particulier et n'est accessible qu'en lecture (`tab.`*`length`*` = 25`; est interdit).

L'exemple ci-dessous déclare une référence sur tableau d'entiers, qui repère un tableau de 20 entiers, ce tableau contenant les valeurs 0 à 19.

```
int[] tab = new int[20];
for (int i=0; i<tab.length; i++) {
     tab[i] = i;
}
```

Méthode `toString()`

Dans **`Object`**, la méthode **`public String`** `toString()` est déclarée ; elle retourne une chaîne de caractères représentant l'objet : sa classe et son adresse. Lorsqu'un objet quelconque est passé comme paramètre à **`System`**`.`*`out`*`.println()`, c'est le résultat de l'appel à `toString()` sur cet objet qui est affiché. Afin de pouvoir afficher simplement l'état des objets du programme, on redéfinit souvent la méthode `toString()` dans les classes qu'on crée. Beaucoup de classes de l'API Java redéfinissent aussi la méthode `toString()`.