Concepts de base de la programmation Java

Nasser Benameur / Tondeur Hervé

Caractéristiques du langage

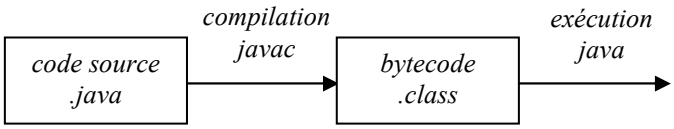
Java est un langage orienté objet, il à une syntaxe proche du C++, le compilateur est fourni avec les packages et les API nécessaire.

Java est est un langage interprété, il est de ce fait portable, il est multithreaded, il n'utilise pas de pointeurs, pas de structures, pas de fonction globales.

La machine virtuelle Java (*JVM*)

Java est un langage *multi-plates-formes* qui permet, selon *Sun Microsystems*, son concepteur, d'écrire une fois pour toute des applications capables de fonctionner dans tous les environnements. Cet objectif est atteint grâce à l'utilisation d'une machine virtuelle Java (JVM) qui exécute les programmes écrits dans ce langage.

Compilation



- compilation du code source : *javac *.java* ;
- exécution sur la JVM : *java MainFile*.

Les classes

Définition

La classe regroupe la définition des *membres de classe*, c'est-à-dire :

- des *méthodes*, les opérations que l'on peut effectuer ;
- des *champs*, les variables que l'on peut traiter ;
- des *constructeurs*, qui permettent de créer des objets ;
- et encore d'autres choses plus particulières.

Plus précisément, une classe peut contenir des *variables (primitives ou objets)*, des *classes internes*, des *méthodes*, des *constructeurs*, et des *finaliseurs*.

La déclaration d'une classe se fait de la façon suivante :

```
[Modificateurs] class NomClasse

        {
                corps de la classe
        }
```

Le nom de la classe doit débuter par une majuscule.

Considérons l'exemple suivant :

```
Class Animal {

   // champs

   boolean vivant ;
   int âge ;

   // constructeurs

   Animal() {
   }

   // méthodes

   void viellit() {
         ++âge ;
   }

   void crie() {
   }

}
```

Les classes *final*

Une classe peut être déclarée *final*, dans un but de sécurité ou d'optimisation. Une classe *final* ne peut être étendue pour créer des sous-classes. Par conséquent, ses méthodes ne peuvent pas être redéfinies et leur accès peut donc se faire sans recherche dynamique.

Les classes internes

Une classe Java peut contenir, outre des primitives, des objets (du moins leurs références) et des définitions de méthodes, des définitions de classe. Nous allons maintenant nous intéresser de plus près à cette possibilité.

- **Plusieurs classes dans un même fichier**

Il arrive fréquemment que certaines classes ne soient utilisées que par une seule autre classe. Considérons l'exemple suivant :

```
Class Animal {

   // champs

   boolean vivant ;
   int âge ;
   Coordonnées position ;

   // constructeurs

   Animal() {
         position = new Coordonnées() ;
         }

   ...

   }

Class Coordonnées {

   // champs
```

```
int x = 0;
int y = 0;

...

}
```

Lors de la compilation du précédent fichier *Animal.java*, le compilateur produit deux fichiers : *Animal.class* et *Coordonnées.class*.

- **Les classes imbriquées ou *static***

Il peut être avantageux dans certains cas de placer la définition d'une classe à l'intérieur d'une autre, lorsque celle-ci concerne uniquement « la classe principale ». Voyons pour notre exemple :

```
Class Animal {

   // champs

   boolean vivant ;
   int âge ;
   Coordonnées position ;

   // classes imbriquées

   static Class Coordonnées {

         // champs

         int x = 0;
         int y = 0;

         ...

      }

   // constructeurs

   Animal() {
         position = new Coordonnées() ;
         }

   ...

   }
```

La définition de la classe *Coordonnées* est maintenant imbriquée dans la classe *Animal*. Par ailleurs, la référence à la classe *Coordonnées* devient *Animal.Coordonnées*. De manière générale, les références à une classe imbriquée en Java se font en utilisant le point comme séparateur.

Lors de la compilation du fichier ci-dessus, *Animal.java*, le compilateur produit deux fichiers : *Animal.class* et *Animal$Coordonnées.class* pour la classe imbriquée.

Quant au chemin d'accès, notons que *mesclasses.Animal* désigne la classe *Animal* dans le package *mesclasse,* tandis que *mesclasses.Animal.Coordonnées* désigne la classe *Coordonnées* imbriquée dans la classe *Animal,* elle-même contenu dans le package *mesclassses*. Ainsi il est possible d'utiliser la directive *import* pour importer les classes imbriquées explicitement ou en bloc :

```
import mesclasses.Animal.Coordonnées ;
```

```
import mesclasses.Animal.* ;
```

Les classes imbriquées peuvent elles-mêmes contenir d'autres classes imbriquées, sans limitation de profondeur, du moins du point de vue de Java.

Un dernier point. La classe *Coordonnées* a été déclarée *static*, ce qui est obligatoire pour toute classe imbriquée. En revanche, les interfaces imbriquées sont automatiquement déclarées *static* et il n'est donc pas nécessaire de les déclarer explicitement comme telles.

Les champs

Définition

L'état représente l'ensemble des variables qui caractérisent une classe ; on parle encore de champs ou de membres. Notons que Java initialise par défaut les variables membres.

Considérons l'exemple suivant :

```
Class Animal {

   // champs

   int âge ;
   static int longévité = 100 ;

}
```

Variables d'instances & Variables *static*

Dans l'exemple ci-dessus, *âge* est une variable d'instance, tandis que *longévité* représente une variable *static*.

Dans cet exemple, nous avons considéré que la *longévité* était une caractéristique commune à tous les animaux, mettons 100 ans ! Il n'est donc pas nécessaire de dupliquer cette information dans chacune des instances de la classe. Nous avons donc choisi de déclarer *longévité* comme une variable *static*. Il en résulte que cette variable appartient à la classe et non à ses instances.

Pour comprendre cette nuance, considérons une instance de la classe *Animal,* appelé *monAnimal*. L'objet *monAnimal* possède sa propre variable *âge*, à laquelle il est possible d'accéder grâce à la syntaxe :

```
monAnimal.âge
```

L'objet *monAnimal* ne possède pas de variable *longévité*. Normalement, *longévité* appartient à la classe *Animal*, et il est possible d'y accéder en utilisant la syntaxe :

```
Animal.longévité
```

Cependant, Java nous permet également d'utiliser la syntaxe :

```
monAnimal.longévité
```

Mais il faut bien comprendre que ces deux expressions font référence à la même variable. On peut utiliser le nom de la variable seul pour y faire référence, uniquement dans la définition de la classe.

Les variables *final*

Une variable déclarée *final* ne peut plus voir sa valeur modifiée. Elle remplit alors le rôle de constante dans d'autres langages. Une variable *final* est le plus souvent utilisée pour encoder des valeurs constantes.

Par exemple, on peut définir la constante *Pi* de la manière suivante :

```
final float pi = 3.14 ;
```

Déclarer une variable *final* offre deux avantages. Le premier concerne la sécurité. En effet, le compilateur refusera toute affectation ultérieure d'une valeur à la variable. Le deuxième avantage concerne l'optimisation du programme. Sachant que la valeur en question ne sera jamais modifié, le compilateur est à même de produire un code plus efficace. En outre, certains calculs préliminaires peuvent être effectués.

Les méthodes

Les méthodes sont les opérations ou les fonctions que l'on peut effectuer sur une classe. On distingue deux types de méthodes :

- les *accesseurs*, qui ne modifient pas l'état et se contente de retourner la valeur d'un champs ;
- les *modificateurs*, qui modifient l'état en effectuant un calcul spécifique.

Une déclaration de méthode est de la forme suivante :

```
[Modificateurs] Type nomMéthode ( paramètres ...)

      {
            corps de la méthode
      }
```

Le nom de la méthode débute par une minuscule ; la coutume veut qu'un accesseur débute par le mot « *get* » et qu'un modificateur débute par le mot « *set* ».

Les méthodes *static*

Les méthodes peuvent également être déclaré *static*. Imaginons que nous souhaitons construire un *accesseur* pour la variable *longévité* (voir exemple précédent). Nous pouvons le faire de la façon suivante :

```
Class Animal {

   // champs

   int âge ;
   static int longévité = 100 ;

   // méthodes

   static int getLongévité() {
         return longévité ;
   }

}
```

La méthode *getLongévité* peut être déclaré *static* car elle ne fait référence qu'à des membres *static* (en l'occurrence, la variable *longévité*). Ce n'est pas une obligation. Le programme fonctionne aussi si la méthode n'est pas déclaré *static*. Dans ce cas, cependant, la méthode est dupliquée chaque fois qu'une instance est créée, ce qui n'est pas très efficace.

Comme dans le cas des variables, les méthodes *static* peuvent être référencées à l'aide du nom de la classe ou du nom de l'instance. On peut utiliser le nom de la méthode seul , uniquement dans la définition de la classe.

Il est important de noter que les méthodes *static* ne peuvent en aucun cas faire référence aux méthodes ou aux variables non *static* de la classe. Elles ne peuvent non plus faire référence à une instance. (La référence *this* ne peut pas être employé dans la méthode *static*.)

Les méthodes *static* ne peuvent pas non plus être redéfinies dans les classes dérivées.

Les méthodes *final*

Les méthodes peuvent également être déclaré *final*, ce qui restreint leur accès d'une toute autre façon. En effet, les méthodes *final* ne peuvent pas être redéfinies dans les classes dérivées. Ce mot clé est utilisé pour s'assurer que la méthode d'instance aura bien le fonctionnement déterminé dans la classe parente. (S'il s'agit d'une méthode *static*, il n'est pas nécessaire de la déclarer *final* car les méthodes *static* ne peuvent jamais être redéfinies.)

Les méthodes *final* permettent également au compilateur d'effectuer certaines optimisations qui accélèrent l'exécution du code. Pour déclarer une méthode *final*, il suffit de placer ce mot clé dans sa déclaration de la façon suivante :

```
final int calcul(int i, int j) {...}
```

Le fait que la méthode soit déclaré *final* n'a rien à voir avec le fait que ces arguments le soient ou non.

Nous reviendrons sur l'utilité des méthodes *final* dans le chapitre concernant le *polymorphisme*, et notamment le concept *early & late binding*.

Les constructeurs

Les constructeurs : création d'objets

Les constructeurs et les initialiseurs sont des éléments très importants car ils déterminent la façon dont les objets Java commencent leur existence. Ces mécanismes, servant à contrôler la création d'instance de classe, sont fondamentalement différents des méthodes.

- **Les constructeurs (*constructor*)**

Les constructeurs sont des méthodes particulières en ce qu'elles portent le même nom que la classe à laquelle elles appartiennent. Elles sont automatiquement exécutées lors de la création d'un objet. Le constructeur par défaut ne possèdent pas d'arguments.

- Les constructeurs n'ont pas de type et ne retournent pas.
- Les constructeurs ne sont pas hérités par les classes dérivées.
- Lorsqu'un constructeur est exécuté, les constructeurs des classes parentes le sont également. C'est *le chaînage des constructeurs*. Plus précisément, si en première instruction le compilateur ne trouvent pas un appel à *this(...)* ou *super(...)*, il rajoute un appel à *super(...)*. L'utilisation de *this(...)* permet de partager du code entre les constructeurs d'une même classe, dont l'un au moins devra faire référence au constructeur de la super-classe.
- Une méthode peut porter le même nom qu'un constructeur, ce qui est toutefois formellement déconseillé.

- **Exemple de constructeurs**

Considérons l'exemple suivant :

```
class Animal {

   // champs

   boolean vivant ;
   int âge ;

   // constructeurs
```

```
    Animal() {
    }

    Animal(int a) {
          âge = a ;
          vivant = true ;
    }

    // méthodes

}
```

Si nous avions donné au paramètre *a* le même nom que celui du champs *âge*, il aurait fallu accéder à celle-ci de la façon suivante :

```
    Animal(int âge) {
          this.âge = âge ;
          vivant = true ;
    }
```

Toutefois, pour plus de clarté, il vaut mieux leur donner des noms différents. Dans le cas de l'initialisation d'une variable d'instance à l'aide d'un paramètre, on utilise souvent pour le nom du paramètre la première (ou les premières) lettre(s) du nom de la variable d'instance.

- **Création d'objets (*object*)**

Tout objet *java* est une *instance* d'une classe. Pour allouer la mémoire nécessaire à cet objet, on utilise l'opérateur *new*, qui lance l'exécution du constructeur.

La création d'un *Animal* se fait à l'aide de l'instruction suivante :

```
    Animal nouvelAnimal = new Animal(3) ;
```

- **Surcharger les constructeurs**

Les constructeurs, tout comme les méthodes, peuvent être surchargés dans le sens où il peut y avoir plusieurs constructeurs dans une même classe, qui possèdent le même nom (celui de la classe). Un constructeur s'identifie de part sa signature qui doit être différente d'avec tous les autres constructeurs.

Supposons que la plupart des instances soient créées avec 0 pour valeur initiale de *âge*. Nous pouvons alors réécrire la classe *Animal* de la façon suivante :

```
class Animal {

    // champs

    boolean vivant ;
    int âge ;

    // constructeurs

    Animal() {
          âge = 0 ;
          vivant = true ;
    }

    Animal(int a) {
          âge = a ;
          vivant = true ;
    }
```

```
    // méthodes

}
```

Ici, les deux constructeurs possèdent des signatures différentes. Le constructeur sans paramètre traite le cas où l'âge vaut 0 à la création de l'instance. Une nouvelle instance peut donc être créée sans indiquer l'âge de la façon suivante :

```
Animal nouvelAnimal = new Animal() ;
```

- **Autorisation d'accès aux constructeurs**

Les constructeurs peuvent également être affectés d'une autorisation d'accès. Un usage fréquent de cette possibilité consiste comme pour les variables, à contrôler leur utilisation, par exemple pour soumettre l'instanciation à certaines conditions.

Initialisation des objets

Il existe en Java trois éléments pouvant servir à l'initialisation :
- *les constructeurs*,
- *les initialiseurs de variables d'instances et statiques*,
- *les initialiseurs d'instances et statiques*.

Nous avons déjà présenté l'initialisation utilisant un constructeur. Voyons les deux autres manières.

- **Les initialiseurs de variables d'instances et statiques**

Considérons la déclaration de variable suivante :

```
int a ;
```

Si cette déclaration se trouve dans une méthode, la variable n'a pas de valeurs. Toute tentative d'y faire référence poduit une erreur de compilation.

En revanche, s'il s'agit d'une *variable d'instance* (dont la déclaration se trouve en dehors de toute méthode), Java l'initialise automatiquement au moment de l'instanciation avec une valeur par défaut. Pour *les variables statiques*, l'initialisation est réalisée une fois pour toute à la première utilisation de la classe.

Les variables de type numérique sont initialisées à *0*. Le type booléen est initialisé à *false*.

Nous pouvons cependant initialiser nous-mêmes les variables de la façon suivante :

```
int a = 1 ;
int b = a*7 ;
float c = (b-c)/3 ;
boolean d = (a < b) ;
```

Les initialiseurs de variables permettent d'effectuer des opérations d'une certaine complexité, mais celle-ci est tout de même limitée. En effet, ils doivent tenir sur une seule ligne. Pour effectuer des opérations plus complexes, il convient d'utiliser les constructeurs ou encore les initialiseurs d'instances.

- **Les initialiseurs d'instances**

Un initialiseur d'instance est tout simplement placé, comme les variables d'instances, à l'extérieur de toute méthode ou constructeur.

Voyons l'exemple suivant :

```
class Exemple {
```

```
    // champs

    int a ;
    int b ;
    float c ;
    boolean d ;

    // initialiseurs

    {
          a = 1 ;
          b = a*7 ;
          c = (b-a)/3 ;
          d = (a < b);
    }

}
```

- **Les initialiseurs statiques**

Un *initialiseur statique* est semblable à un *initialiseur d'instance*, mais il est précédé du mot *static*. Considérons l'exemple suivant :

```
class Voiture {

    // champs

    static int capacité ;

    // initialiseurs

    static {
          capacité = 80;
          System.out.println("La variable vient d'être initialisée.\n") ;
    }

    // constructeurs

    Voiture() {
    }

    // méthodes

    static int getCapacité() {
          return capacité;
          }

}
```

L'initialiseur statique est exécuté au premier chargement de la classe, que ce soit pour utiliser un membre statique, `Voiture.getCapacité()` ou pour l'instancier, `Voiture maVoiture = new Voiture()`.

Les membres statiques (ici la variable `capacité`) doivent être déclarés avant l'initialiseur. Il est possible de placer plusieurs initialiseurs statiques, où l'on souhaite dans la classe. Ils seront tous exécutés au premier chargement de celle-ci, dans l'ordre où ils apparaissent.

La destruction des objets (*garbage collector*)

Avec certains langages, le programmeur doit s'occuper lui-même de libérer la mémoire en supprimant les objets devenus inutiles. Avec Java, le problème est résolu de façon très simple : un programme, appelé *garbage*

collector, ce qui signifie littéralement « ramasseur d'ordures », est exécuté automatiquement dès que la mémoire disponible devient inférieure à un certain seuil. De cette façon, aucun objet inutilisé n'encombrera la mémoire.

Le concept de l'héritage

Hiérarchie des classes

De plus chaque classe *dérive* d'une classe de niveau supérieur, appelée *sur-classe*. Cela est vrai pour toutes les classes sauf une. Il s'agit de la classe *Object*, qui est l'ancêtre de toutes les classes.

Toute instance d'une classe est un objet du type correspondant, mais aussi du type de toutes ses classes ancêtres.

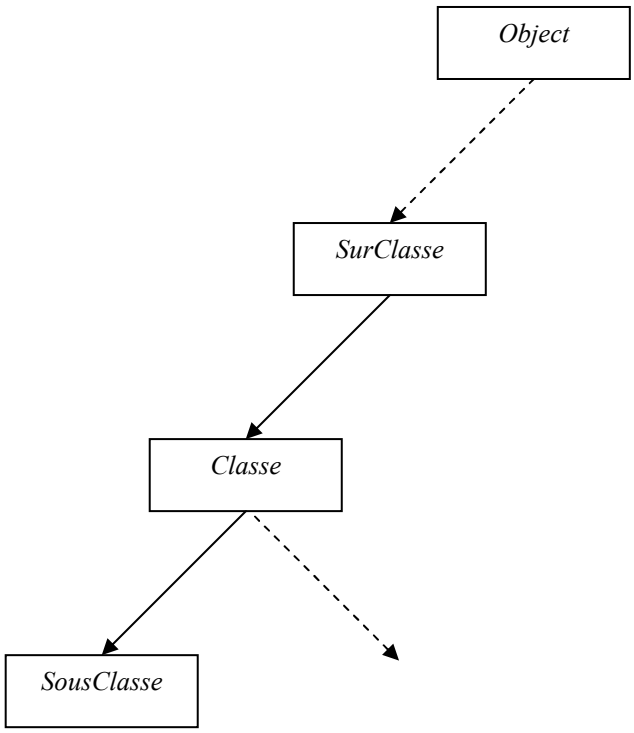
SousClasse hérite de toutes les caractéristiques de *Classe*, et donc, par transitivité, de *SurClasse*.

Une *classe* est toujours construite à partir d'une autre classe dont elle est *dérivée*. Une classe dérivée est une *sous-classe* d'une *sur-classe*.

- **Extends**

Lorsque le paramètre *extends* est omis, la classe déclarée est une sous classe de l'objet *Objet*.

- **Référence à la classe parente**



Appel du constructeur de la SurClasse

Il faut utiliser le mot réservé super()

Exemple :

```
Class Point
{int x,y ;
Point(int x, int y) {…}
Void affiche(){…}
```

```
}

class Point3D extends Point
{int z ;
Point3D(int a,int b,int c)
{super(a,b) ;
z=c ;}

}
```

Redéfinition des champs et des méthodes

- **Redéfinition des méthodes**

Une deuxième déclaration d'une méthode dans une classe dérivée remplace la première. Voyons un exemple avec la méthode *crie( )* redéfinie dans la classe *Chien* dérivée de la classe *Animal*.

```
Class Animal {

   // champs

   // méthodes

   void crie() {
   }

}

Class Chien extends Animal {

   // champs

   // méthodes

   void crie() {
         System.out.println("Ouah-Ouah !") ;
   }

}
```

La surchage

- **Surcharger les méthodes**

Une méthode est dite surchargée si elle permet plusieurs passages de paramètres différents.

Accessibilité

En Java, il existe quatre catégories d'autorisations d'accès, spécifiés par les modificateurs suivants : *private, protected, public*. La quatrième catégorie correspond à l'absence de modificateur.

Nous allons les présenter en partant du moins restrictif jusqu'au plus restrictif.

- **public**

Les classes, les interfaces, les variables (primitives ou objets) et les méthodes peuvent être déclarées *public*.

Les éléments *public* peuvent être utilisés par n'importe qui sans restriction ; il est accessible à l'extérieur de la classe.

- **protected**

Cette autorisation s'applique uniquement aux membres de classes, c'est-à-dire aux variables (objets ou primitives), aux méthodes et aux classes internes. Les classes qui ne sont pas membre d'une autre classe ne peuvent pas être déclarées *protected*.

Dans ce cas, l'accès en est réservé aux méthodes des classes appartenant au même *package*, aux classes dérivées de ces classes, ainsi qu'aux classes appartenant aux mêmes *packages* que les classes dérivées. Plus simplement, on retiendra qu'un élément déclaré *protected* n'est visible que dans la classe où il est défini et dans ses sous-classes.

- **friendly**

L'autorisation par défaut s'applique aux classes, interfaces, variables et méthodes.

Les éléments qui disposent de cette autorisation sont accessibles à toute les méthodes des classes du même package. Les classes dérivées ne peuvent donc y accéder que si elles sont explicitement déclarées dans le même package.

Rappelons que les classes n'appartenant pas explicitement à un package appartiennent automatiquement au package par défaut. Toute classe sans indication de package dispose donc de l'autorisation d'accès à toutes les classes se trouvant dans le même cas.

- **private**

L'autorisation *private* est la plus restrictive. Elle s'applique aux membres d'une classe (variables, méthodes, classes internes).

Les éléments déclarés *private* ne sont accessibles que depuis la classe qui les contient ; il n'est visible que dans la classe où il est défini.

Ce type d'autorisation est souvent employé pour les variables qui ne doivent être lues ou modifiées qu'à l'aide d'un *accesseur* ou d'un *modificateur*. Les accesseurs et les modificateurs, de leur côté, sont déclarés *public*, afin que tout le monde puisse utiliser la classe.

Tableau des modificateurs

Package 1

Class A

{int a ; public int b ;

protected int c ;

private int d ;

privateprotected int e ;

}

class B extends A

{}

class C{}

Package2

Class D extends A{}

Class E {}

	Acces pour toutes les classes du package	Vu par tous	Pour les héritiers et les classes du même package	Uniquement dans la classe	Uniquement pour les héritiers
A partir de	**Friendly**	**public**	**protected**	**private**	**Private protected**
B extends A	**OUI**	**OUI**	**OUI**	**NON**	**OUI**
C	**OUI**	**OUI**	**OUI**	**NON**	**NON**
D extends A	**NON**	**OUI**	**OUI**	**NON**	**OUI**
E	**NON**	**OUI**	**NON**	**NON**	**NON**

Les classes abstraites, les interfaces, le polymorphisme

Le mot clé *abstract*

- **Méthodes et classes abstraites**

Une méthode déclarée *abstract* ne peut être exécutée. En fait, elle n'a pas d'existence réelle. Sa déclaration indique simplement que les classes dérivées doivent la redéfinir.

Les méthodes *abstract* présentent les particularités suivantes :

- Une classe qui contient une méthode *abstract* doit être déclarée *abstract*.
- Une classe *abstract* ne peut pas être instanciée.
- Une classe peut être déclarée *abstract*, même si elle ne comporte pas de méthodes *abstract.*
- Pour pouvoir être instanciée, une sous-classe d'une classe *abstract* doit redéfinir toute les méthodes *abstract* de la classe parente.
- Si une des méthodes n'est pas redéfinie de façon concrète, la sous-classe est elle-même *abstract* et doit être déclarée explicitement comme telle.
- Les méthodes *abstract* n'ont pas d'implémentation. Leur déclaration doit être suivie d'un point-virgule.

Ainsi dans l'exemple précédent la méthode *crie( )* de la classe *Animal* aurait pu être déclarée *abstract*, ce qui signifie que tout *Animal* doit être capable de crier, mais que le cri d'un animal est une notion abstraite. La méthode ainsi définie indique qu'une sous-classe devra définir la méthode de façon concrète.

```
abstract class Animal {

   // champs

   // méthodes

   abstract void crie() ;
}

class Chien extends Animal {

   // champs
```

```
    // méthodes

    void crie() {
          System.out.println("Ouah-Ouah !") ;
    }

}

class Chat extends Animal {

    // champs

    // méthodes

    void crie() {
          System.out.println("Miaou-Miaou !") ;
    }

}
```

De cette façon, il n'est plus possible de créer un animal en instanciant la classe *Animal*. En revanche, grâce à la déclaration de la méthode *abstract crie( )* dans la classe *Animal*, il est possible de faire crier un animal sans savoir s'il s'agit d'un chien ou d'un chat, en considérant les instances de Chien ou de Chat comme des instances de la classe parente.

```
Animal animal1 = new Chien() ;
Animal animal2 = new Chat() ;

animal0.crie() ;
animal1.crie() ;
```

Le premier animal crie "Ouah-Ouah !" ; le second, "Miaou-Miaou !".

Les interfaces

Une classe peut contenir des méthodes *abstract* et des méthodes non *abstract*. Cependant, il existe une catégorie particulière de classes qui ne contient que des méthodes *abstract*. Il s'agit des interfaces. Les interfaces sont toujours *abstract*, sans qu'il soit nécessaire de l'indiquer explicitement. De la même façon, il n'est pas nécessaire de déclarer leurs méthodes *abstract*.

Les interfaces obéissent par ailleurs à certaines règles supplémentaires.

- Elles ne peuvent contenir que des variables *static* et *final*.

- Elles peuvent être étendues comme les autres classes, avec une différence majeure : une interface peut dériver de plusieurs autres interfaces. En revanche, une classe ne peut pas dériver uniquement d'une ou de plusieurs interfaces. Une classe dérive toujours d'une autre classe, et peut dériver, en plus, d'une ou plusieurs interfaces.

Interface Point {…}

Class Point3D implements Point, implements …., implements… {…}

Casting

- **Sur-casting**

Un objet peut être considéré comme appartenant à sa classe ou à une classe parente selon le besoin, et cela de façon dynamique. Nous rappelons ici que toutes classes dérive de la classe *Object,* qui est un type commun à tous les objets. En d'autres termes, le lien entre une classe et une instance n'est pas unique et statique. Au

contraire, il est établi de façon dynamique, au moment où l'objet est utilisé. C'est la première manifestation du polymorphisme !

Le sur-casting est effectué de façon automatique par Java lorsque cela est nécessaire. On dit qu'il est implicite. On peut l'expliciter pour plus de clarté, en utilisant l'opérateur de casting :

```
Chien chien = new Chien() ;
Animal animal = (Animal)chien ;
```

Après cette opération, ni le handle *chien,* ni l'objet correspondant ne sont modifiés. Les handles ne peuvent jamais être redéfinie dans le courant de leur existence. Seule la nature du lien qui lie l'objet aux handles change en fonction de la nature des handles.

Le sur-casting est un peu moins explicite, lorsqu'on affecte un objet à un handle de type différent. Par exemple :

```
Animal animal = new Chien() ;
```

Polymorphisme

- **Utilisation du sur-casting**

Considérons l'exemple suivant, qui reprend les définitions précédentes des classes *Animal, Chien,* et *Chat* et illustre une première sorte de polymorphisme avec sur-casting implicite sur la méthode *crie( )*.

```
public class Main {

   // méthodes

   public static void main(String[] argv) {
         Chien chien = new Chien() ;
         Chat chat = new Chat() ;
         crie(chien) ;
         crie(chat) ;

   void crie(Animal animal) {
         animal.crie() ;
   }
}
```

Le premier animal crie "Ouah-Ouah !" ; le second, "Miaou-Miaou !".

- ***Late-binding***

Il existe un moyen d'éviter le sous-casting explicite en Java, appelé *late-binding*. Cette technique fondamentale du polymorphisme permet de déterminer dynamiquement quelle méthode doit être appelée.

Dans la plupart des langages, lorsque le compilateur rencontre un appel de méthode, il doit être à même de savoir exactement de quelle méthode il s'agit. Le lien entre l'appel et la méthode est alors établi à la compilation. Cette technique est appelée *early binding* (liaison précoce). Java utilise cette technique pour les appels de méthodes déclarées *final*. Elle a l'avantage de permettre certaines optimisations.

En revanche, pour les méthodes qui ne sont pas *final*, Java utilise la technique du *late binding* (liaison tardive). Dans ce cas, le compilateur n'établit le lien entre l'appel et la méthode qu'au moment de l'exécution du programme. Ce lien est établi avec la version la plus spécifique de la méthode et doit être différencié du concept *abstract*. Considérons l'exemple suivant pour s'en convaincre.

```
class Animal {

   // méthodes
```

```
      void crie() {
      }
   }

   class Chien extends Animal {

      // méthodes

      void crie() {
            System.out.println("Ouah-Ouah !") ;
      }

   }

   class Chat extends Animal {

      // méthodes

      void crie() {
            System.out.println("Miaou-Miaou !") ;
      }
   }

   public class Main {

      // méthodes

      public static void main(String[] argv) {
            Chien chien = new Chien() ;
            Chat chat = new Chat() ;
            crie(chien) ;
            crie(chat) ;

      void crie(Animal animal) {
            animal.crie() ;
      }
}
```

Le premier animal crie "Ouah-Ouah !" ; le second, "Miaou-Miaou !". La méthode *crie* appelé dans la méthode *crie* de la classe *Main* est bien la plus spécifique, celle de *Chien* ou de *Chat* et non celle de *Animal* !

- ***Polymorphisme***

Le programme ci-dessous illustre le concept du polymorphisme. La classe *Animal* utilise la méthode abstraite *qui* pour définir la méthode *printQui* de manière plus ou moins abstraite. Cela entraîne une factorisation du code.

```
   abstract class Animal {

      // méthodes

      abstract String qui() ;

      void printQui() {
            System.out.println("cet animal est un " + qui()) ;
      }
   }

   class Chien extends Animal {

      // méthodes
```

```
    String qui() {
          return "chien" ;
    }
}

class Chat extends Animal {

    // méthodes

    String qui() {
          return "chat" ;
    }
}
```

L'implémentation de la méthode *qui* est relié à l'appel, uniquement au moment de l'exécution, en fonction du type de l'objet appelant et non celui du handle !
C'est-à-dire,

```
Animal animal1 = new Chien() ;
Animal animal2 = new Chat() ;
animal1.printQui() ;
animal2.printQui() ;
```

donne comme résultat :

```
cet animal est un chien
cet animal est un chat
```

Spécificités du langage

Programme principal : la méthode *main*

(…)

Cette méthode doit impérativement être déclaré public. Ainsi la classe contenant la méthode *main( )*, le programme principal, doit être *public*. Rappelons ici qu'un fichier contenant un programme Java ne peut contenir qu'une seule définition de classe déclarée *public*. De plus le fichier doit porter le même nom que la classe, avec l'extension *.java*.

Class MonProg
{

Public static void main(String argv[])
{…}

}

Passage des arguments

Transmettre un argument à une application

Un argument se transmet à une application à l'exécution :

```
// sans transmission d'argument
java NomDuProgramme
```

```
// avec transmission d'argument
java NomDuProgramme PremierArgument SecondArgument TroisièmeArgument EtQuaetera
```

On note que les arguments sont séparés d'un espace. Dans le cas d'un argument qui comprend plusieurs mots qui doivent être séparés, on utilise les guillemets : " Federico Garcia Lorca " est considéré comme 1 seul argument.

Récupérer un argument à partir d'une application

On doit créer une boucle for qui parcourt le programme à la recherche d'arguments :

```
//partie théorique
for ( int i = 0 ; i < arguments.length ; i++ ) {–}

//partie pratique
class MesArguments {
public static void main(String args[ ]) {
for ( int i = 0 ; i < arguments.length ; i++ ) {
System.out.println ( iyArguments iy + I + iy : iy + arguments[ I ];
}
}
}
```

Le petit exemple présenté ci dessus affiche les arguments avec leur n°. Vous pouvez compiler ce programme, il fonctionne.

Transmettre un argument à une applet

On transmet un argument à une applet par l'intermédiaire du code HTML :

```
< PARAM NAME = * VALUE = * >
< PARAM NAME = font VALUE = * >
```

Les * symbolisent les valeurs que vous devez spécifier, la première le nom du paramètre et la seconde sa valeur..

Récupérer un argument à partir d'une applet

Pour récupérer à partir d'une applet les arguments transmis par le HTML, on utilise la méthode *getParameter()*.

- TypeDeDonnée NomDeLaVariable = getParameter (" nomDuParamètre ") ;
- String theFont = getParameter(" font ") ;

Package

Les packages

- **Les packages accessibles par défaut**

La bibliothèque standard de Java est distribuée dans un certain nombre de packages, y compris java.lang, java.util, java.net, etc….Les packages standard de Java constituent des packages hiérarchiques. Tout comme les répertoires d'un disque dur, les packages peuvent être organisés suivant plusieurs niveaux d'imbrication. Tous les packages standard de Java se trouvent au sein des hiérarchies de packages java et javax.

- **L'instruction *package***

Une classe peut utiliser toutes les classes de son propre package et toutes les classes publiques des autres packages.
Vous pouvez accéder aux classes publiques d'un autre package de deux façons.

La première façon consiste simplement à ajouter le nom complet du package devant chaque nom de classes.

Exemple :
Java.util.Date today=new java.util.Date() ;

La seconde est d'utiliser l'instruction import

- **L'instruction *import***

Vous pouvez importer une classe spécifique ou l'ensemble d'un package. En plaçant l'instruction import en tête de vos fichiers source.
Exemple :

Import java.util.* ;

Date today=new Date() ;

Sans le préfixe du package, il est également possible d'importer une classe spécifique d'un package.

Import java.util.Date ;

Les tableaux

Les tableaux sont des objets, il faut utiliser l'opérateur NEW :

Exemple de déclaration et d'utilisation :

Int T[], int[]T ;
T=new int[10] ;

Ou int []T=new int[10] ;

Il existe une méthode length qui donne la longueur du tableau.

Int longueur=int.length nb T.length est interdit d'utilisation.

Les tableaux multidimensionnels.

Int M[][]
M=new int[10][20] ;

On peut initialiser un tableau lors de sa déclaration sans utiliser new

Exemple int[] T={3,4,2,1} ici new est inutile et implicite.

Les chaînes de caractères

Les chaînes de caractères sont des objets également.

Il existe deux classes :
String Chaîne constante (Non modifiable)
StringBuffer (Chaîne que l'on peut modifier)

Constructeur de String :

String ch1=new String(« Bonjour ») ;

String ch1= »Bonjour » ;

Quelques méthode sur la classe String (méthode statique)

Static String valueof(int i) ;
Renvoie la chaîne du type donnée.

Exemple : String valueof(12) => « 12 »

Boolean equals (String s) ;
Compare deux chaine.

String concat(String s) ;
Concaténation de deux chaînes.

Int length() ;
Retourne la longueur de la chaîne de caractères.

Int indexof (int c) ;
Renvoie un entier correspondant à l'occurrence du caractère.

Char charAt(int i) ;
Renvoie le caractére de l'indice i.

Quelques méthode de la classe StringBuffer

Les constructeurs :

StringBuffer()
StringBuffer(int taille)
StringBuffer(String s)

StringBuffer append(String s) ; Ajouter une chaîne de caractères
StringBuffer append (char c) ; Ajouter un caractère

Int length() ; renvoie la longueur de la chaîne

String toString() ; Renvoie une chaîne constante de l'objet courant.

Les classes utilitaires

Vector :

C'est un tableau dynamique

Vector v=new Vector() ;

Il peut contenir un nombre quelconque d'objets, mais pas de type simple.

Quelques méthodes :

Void AddElement(object) ;

Void InsertElementAt(object,int) ;

Void SetElementAt(object,int) ;

Void RemoveElement(object) ;

Void RemoveElementAt(int) ;

Void RemoveAllElements() ;

Boolean Contains(object) ;

Object ElementAt(int);

Int IndexOf(object) ;

Int size() ;

Enumeration elements() ;

Exemple d'utilisation :

```
Vector T=new Vector() ;
For (Enumeration e=T.Elements() ;e.HasMoreElements() ;)
System.out.println(e.nextElement()) ;
```

Enumarations :

Représente une liste d'éléments, c'est une interface et non pas une classe, elle possède deux méthodes.

Enumeration E=new Enumeration() ;

Boolean HasMoreElements() ;

Object nextElement() ;

Hashtable :

Table de stockage permettant des recherches rapide, c'est un tableau à deux entrées.

Hashtable H=new Hashtable() ;

Object Put(clé,donnée) ;

Object Get(clé) ;

Object Remove(clé) ;

Boolean ContainsKey(clé) ;

EnumeratEnumeration

Enumeration Elements() ;

Les Applets

Une applet est un programme qui s'éxécute sur une page Web.

Nb : Commentaire à ajouter dans le fichier source java pour éviter de faire une page HTML, ce qui permet d'utiliser l'utilitaire appletviewer.exe pour éxécuter notre applet.

Exemple : appletviewer monapplet.java

Pour cela il faut introduire en commentaire la ligne suivante :

//<applet code= »MonApplet.class » height=400 width=400></applet>

Il n'y a pas de fonction main dans une applet.

Il y a par contre les méthodes suivantes à surcharger obligatoirement :

Void init() {…} Exécuté au début du programme.

Void start(){…} Elle s'éxécute après init à chaque fois que l'appel à été stoppé et relancé.

Void stop(){…} S'éxécute quand l'applet est stoppée.

Void destroy(){…} S'éxécute à la destruction du programme.

Void paint(){…} Permet de dessiner sur la feuille de l'applet.

Système de coordonnées

On place des éléments dans le cadre d'une applet en définissant leur position par des coordonnées. Le repère a pour origine le point le plus en haut et le plus à gauche du cadre.

Segments de droites

Pour tracer le segment de droite [AB], on utilise la commande *DrawLine(xA,yA,xB,yB*).

Rectangles

Il existe 4 types de rectangles : rectangle aux angles droits
vide et plein, rectangle aux angles arrondis vide ou plein.

```
drawRect(x,y,L,h) ;
fillRect(x,y,L,h) ;
drawRoundRect(x,y,L,h, ., .) ;
fillRoundRect(x,y,L,h, ., .);
```

(x,y) sont les coordonnées de l'angle supérieur gauche du rectangle.
L et h sont la largeur et la hauteur du rectangle.
. et . sont les valeurs de largeur et de hauteur de l'angle.

Polygones

Les polygones sont des ensembles de segments de droites joints. Il y a 2 types de polygones : les vides et les pleins (comme les rectangles). On peut les construire de 2 manières : soit en définissant manuellement tous les segments qui définissent le polygone soit en créant une matrice de coordonnées x et une autre de y.

- polygon(int[ ], int[ ], int) ;

Le 3° entier représente le nombre de points du poly gone. Un petit exemple (pas tout à fait fonctionnel) :

- int x[ ] = { _0, 20, 30, 40, 50} ;
- int y[ ] = {50, 60, 70, 80, 90} ;
- int points = x.length ;
- Polygon poly = new Polygon(x, y, points);
- *.drawPolygon(poly);

L'astérisque peut changer de valeurs, on trouve souvent *screen* ou g. Nous verrons cela plus tard.

Ovales

Soit (x,y) les coordonnées du point supérieur gauche,
- drawOval(x,x,largeur,hauteur) ;
- fillOval ;

Arcs de cercle

Là, il faut un schéma–
- drawArc(x,y,largeur,hauteur,angle,nombreDeDegrès) ;
- fillArc() ;

(x,y) les coordonnées du point haut gauche
angle = l'angle à partir duquel l'arc commence (ici, 90°)
nombreDeDegrès = nombre de degrés que parcourt l'arc (ici, 180°)

Copier/Couper

Ce n'est pas vraiment un copier/coller mais on peut s'en servir comme tel.
Ces commandes copient ou coupent une zone rectangulaire.
- CopyArea()
- ClearRect()

Texte et Polices

Pour écrire du texte, on utilise la commande drawString(texte, x, y).
On peut mettre une police en état normal, italique, ou gras :
- Font.PLAIN ; //état normal
- Font.BOLD ; // état gras
- Font.ITALIC ; // état italique

Un petit exemple (encore pas tout à fait fonctionnel) :
- Font f = new Font (" TimesRoman ", Font.BOLD + Font.ITALIC, 24) ; // 24 est la taille
- Screen.setFont(f) ;
- Screen.drawString(i_Bonjour !Ir, 30, 30);

Pour obtenir des informations sur du texte:
- stringWidth(String) // largeur totale de la chaîne
- charWidth(char) // largeur du caractère
- getHeight // hauteur de la police

this.getSize() ; renvoie la dimension de l'objet courant

Dimension d=this.getsize() ;

On peut utiliser d.height ; et d.width par exemple…

En Java 2, Java2D remplace les classe Graphics de Java 1. Java2D est plus performant et gère les couleurs.

Convertir un objet en 2D

Pour créer des graphiques avec Java 2D, on déclare la méthode paint() de l'applet comme avant, mais dans les instructions qui accompagnent cette méthode, on doit convertir l'objet Graphics en objet Graphics2D.

```
public void paint ( Graphics screen ) {
Graphics2D screen2D = (Graphics2D) screen;
// suite des instructions
}
```

Couleurs

Pour colorer quelque chose, on doit d'abord créer un objet color et ensuite attribuer cet objet à la chose que l'on veut colorer.

```
Color c1 = new Color(0.807F, 1F, 0F) ; // nombres à virgule flottante
Color c2 = new Color(255, 204, 102); // nombres normaux
screen2D.setColor(Color.yellow); // couleur basique prédéfinie
```

La 3 ème instruction indique que toutes les opérations de dessin se feront dans la couleur définie (ici, le jaune).
Pour modifier les couleurs de fond d'écran (background) et de dessin (foreground), on utilise ces méthodes :

```
setBackground(Color.white) ;
setForeground(Color.black);
```

Voici un tableau avec les principales couleurs:
Nom de la couleur en Java Valeur RVB En français

black (0,0,0) Noir
blue (0,0,255) Bleu
cyan (0,255,255) Cyan (bleu primaire)
darkGray (64,64,64) Gris foncé
gray (128,128,128) Gris
green (0,255,0) Vert
lightGray (192,192,192) Gris clair
magenta (255,0,255) Magenta (rouge primaire)
orange (255,200,0) Orangé
pink (255,175,175) Rose
red (255,0,0) Rouge (scarlet)
white (255,255,255) Blanc
yellow (255,255,0) Jaune

Les couleurs RVB représentent les valeurs entre 0 et 255 de rouge, vert et bleu.
En fonction de leur concentration, les couleurs se nuancent. La capture d™écran de Macromedia Freehand illustre le choix d™une couleur Camel (189,156,82)..

Dégradés de couleurs

• GradientPaint(x1,y1,Color1,x2,y2,Color2,boolean) ;
x1,x2 et Color1 représentent les coordonnées et la couleur du point de départ du dégradé.
x2,y2 et Color2 représentent les coordonnées et la couleur du point d™arrivée du dégradé.
Boolean est true si le dégradé est cyclique, sinon, il n'est pas nécessaire de le préciser.

Définir un type de trait

Traits :
1) CAP_BUT
2) CAP_SQUARE
3) CAP_ROUND

Jointures
1) JOIN_MITER
2) JOIN_ROUND

3) JOIN_BEVEL

Vous remarquerez qu'il existe une différence de longueur entre les traits 1 et 2. C'est l'unique différence entre CAP_BUT (le classique) et CAP_SQUARE.

Pour créer un trait, il faut utiliser la méthode *setStroke()* de l'objet *BasicStroke()*. Elle admet 3 arguments :
Valeur *float* pour la largeur de la ligne (en pixel).
Valeur *int* pour le style d'ornement, extrémité de ligne.
Valeur *int* pour le style d'ornement, jonction entre 2 lignes.

- BasicStroke pen = BasicStroke (2.0f,BasicStroke.CAP_BATT, BasicStroke.JOIN_ROUND);
- screen2D.setStroke(pen);

Créer des objets à dessiner

Les arguments sont les mêmes que pour des objets.
Tous ces objets font partie de *java.awt.geom*

Lignes :
Lines2D.Float ligne = new Line2D.Float() ;
Rectangles:
Rectangle2D.Float rectangle = new Rectangle2D.Float();
Ellipses:
Ellipse2D.Float ellipse = new Ellipse2D.Float();
Arcs:
Arcs2D.Float = new Arc2D.Float(—.Arc2D.OPEN);
1) Arc2D.OPEN
2) Arc2D.CHORD
3) Arc2D.PIE.Polygones:

- GeneralPath polly = new GeneralPath(); // création d™un i™cheminlt encore invideln
- polly.moveTo(5f,0f); // premier trait
- polly.moveTo(205f,0f); // second trait
- polly.moveTo(5f,90f); // troisième trait
- polly.closePath(); // on referme automatiquement par un quatrième iuplus court possiblelu trait

Les threads

Les *threads* (en français processus indépendants) sont des mécanismes importants du langage Java. Ils permettent d'exécuter plusieurs programmes indépendants les uns des autres. Ceci permet une exécution parallèle de différentes tâches de façon autonome.

Un *thread* réagit aux différentes méthodes suivantes :

- *destroy( )* : arrêt brutal du *thread* ;
- *interrupt( )* permet d'interrompre les différentes méthodes d'attente en appelant une exception ;
- *sleep( )* met en veille de *thread* ;
- *stop( )* : arrêt non brutal du *thread* ;
- *suspend( )* : arrêt d'un *thread* en se gardant la possibilité de le redémarrer par la méthode *resume( ) ;*
- *wait( )* met le *thread* en attente ;
- *yield( )* donne le contrôle au schéduleur.

La méthode *sleep( )* est souvent employée dans les animations, elle permet de mettre des temporisations d'attente entre deux séquences d'image par exemple.

Les exceptions (*exception*)

try–catch

try–catch peut être expliqué comme : « essaye ce bout de code (try), si une exception survient, attrape-la (catch) et exécute le code de remplacement (s'il y en a un) »

```
try {
//code susceptible de produire des exceptions
} catch (Exception e) {
// code de remplacement, se limite souvent à un System.out.println(i.Une erreur : ir+e);
}
```

Exception est la super-classe de toutes les exceptions, elle capte toutes les exceptions. Elle fonctionne dans tous les cas, cependant il est préférable de bien canaliser L'exception en spécifiant le type d'exception le plus précis possible (voir le schéma ci contre). On doit spécifier un nom à cette exception créée, ici c'est e, on choisit généralement e ou ex. Vous pouvez en raison de la portée de variables spécifier le même nom pour toutes vos exceptions (ce nom n'est reconnu qu'à l'intérieur de catch).

Finally

La clause *finally* permet d'exécuter le code qu'elle renferme quoiqu'il arrive. Qu'il y ait une exception ou pas, les instructions de la clause finally sont exécutées. Elle fonctionne également avec try.

```
try {
// instructions à essayer de réaliser
return ;
} finally {
// instructions à réaliser absolument
}
return ;
```

Déclarer des méthodes susceptibles de générer des Exceptions

On utilise la clause throws dans la déclaration de méthode. Exemples :

```
Public boolean myMethod (xxx) throws AnException {–}
Public boolean myMethod (xxx) throws AnException, ASecondException, AThirdException
{–}
Public void myMethod() throws IOException {–}.
```

Générer des exceptions

Ceci sert à faire croire au programme qu'une exception d'un certain type est apparue.

- NotInServiceException() nis = new NotInServiceException("Exception : DataBase out of use");
- throw nis;

On note qu'il s'agit de throw et pas de throws.

Créer des exceptions

Dans des programmes complexes, il se peut que les exceptions standard de Java ne soient pas suffisantes et que vous ayez par conséquent besoin de créer vos propres définitions. Dans ce cas, vos exceptions devront hériter d'une exception plus haute dans la hiérarchie.

```
public class SunSpotException extends Exception {
public SunSpotException () {}
public SunSpotException(String msg) {
super(msg);
}
}
```

Flux d'entrée / Flux de sortie

La création d'un flux de base se fait en 2 étapes : création d'un flux de *support* d'entrée ou de sortie, création d'un flux adapté à l'opération que l'on désire obtenir.

```
// flux de support (valable dans tous les cas)
FileInputStream fis = new FileInputStream(“ fichier.extension “) ;
// flux particulier (exemple)
*InputStream zis = new *InputStream(fis);
```

On crée un flux de support fis grâce à FileInputStream, ce flux (ici, d'entrée) travaille avec le fichier fichier.extension (exemple : Readme.txt), à ce flux, on associe un flux particulier. L'étoile représente tous les types de flux : ZipInputStream, DataInpuStream, GzipInputStream …
Ici sont présentés des flux d'entrée, pour les flux de sortie, on renplace 'Input' par 'Output'.

Exemple concret :

```
FileOutputStream fos = new FileOutputStream(“fichier.txt”);
ZipOutputStream zout = new ZipOutputStream(fos);
```

Classes de flux d'entrée/sortie

1) Entrée
Public int available ( ) throws IOException
Pernet de retourner le nombre d'octets que le flux peut lire. (rarement nécessaire)
Public long skip (long n) throws IOException
Permet d'ignorer les n prochains caractères. (rarement nécessaire)

2) Sortie
Public void flush ( )
Vide immédiatement le contenu d'un flux (vide le tampon et distribue ce qu'il contient). (utile).Public void close ( ) throws IOException

Permet de fermer un flux. Même si vous ne fermez pas un flux, aucune erreur ne sera (normalement) provoquée. Cependant, chaque flux devrait être fermé.

Mise en mémoire tampon

La mise ne mémoire tampon d'un flux permet d'accélérer le performances, voici la procédure à adopter :

```
// création du support
FileInputStream fis = new FileInputStream( “ fichier.txt “);
// création du flux de mise en tampon
BufferedInputStream bis = new BufferedInputStream(fis);
// on joint le BufferedInputStream à un flux particulier, par exemple:
DataInputStream dis = new DataInputStream(bis) ;
```

Readers/Writers

Les readers et writers sont un sujet assez long à traiter. Il serait hors-sujet de les traiter dans leur intégralité dans ce chapitre présentant les concepts clé de I/O. Nous allons donc voir uniquement une partie basique.

Les Readers et Writers permettent de lire et d'écrire des données dans des fichiers. Ils sont très utiles pour travailler avec des fichiers de texte. Ils sont utilisés en remplacement des méthodes FileInputStream et

FileOutputStream. Par exemple, pour utiliser un FileWriter et un FileReader :

```
// FileWriter:
// déterminer le texte à écrire dans le fichier en le prenant, par exemple, dans un TextArea
String texte = new String(jTextArea1.getText());
// ouvrir le FileWriter avec pour argument le nom du fichier de sortie
FileWriter lu = new FileWriter("fichier.txt");
// procédure de mise en cache
BufferedWriter out = new BufferedWriter(lu);
// écrire dans le FileWriter les informations du String texte
out.write(texte);
// fermer le flux
out.close();

// FileReader :
//ouvre un FileReader
FileReader fr = new FileReader(‘’ fichier.txt ‘’) ;
// procédure de lecture des caractères…
While (true) {
// lire les caractères
int i = fr.read();
// -1 représente le moment où il n’y a plus de caractères à lire, le boucle while doit alors s’
// arrêter
if (i == -1) break;
}
```