Programmation en Java

Alexandre Meslé

15 juillet 2009

Table des matières

Chapitre 1

Notes de cours

1.1 Introduction

1.1.1 Hello World !

Copiez le code ci-dessous dans un fichier que vous enregistrerez sous le nom `HelloWorld.java`.

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

Ensuite, exécutez la commande

```
javac HelloWorld.java
```

L'exécution de cette commande doit normalement faire un apparaître un fichier `HelloWorld.class`. Saisissez ensuite la commande :

```
java HelloWorld
```

Théoriquement, ce programme devrait afficher

```
Hello World
```

1.1.2 Formats de fichiers

Les programmes java contiennent ce que l'on appelle des **classes**. Pour le moment nous ne mettrons qu'une seule classe par fichier et nous donnerons au fichier le même nom que la classe qu'il contient. Les fichiers sources portent l'extension `.java` tandis que les programmes compilés portent l'extension `.class`. Le compilateur que nous avons invoqué en ligne de commande avec l'instruction `javac`, a généré le fichier `.class` correspondant au fichier source passé en argument.

1.1.3 Machine virtuelle

Le fichier `.class` n'est pas un exécutable, il contient ce que l'on appelle du **pseudo-code**. Le pseudo-code n'est pas exécutable directement sur la plate-forme (systèle d'exploitation) pour laquelle il a été compilé, il est nécessaire de passer par un logiciel appelé une **machine virtuelle**. La machine virtuelle lit le pseudo-code et l'intéprète (i.e. l'exécute). Le grand avantage de Java est que le pseudo code ne dépend pas de la plate-forme mais de la machine virtuelle. Un programme compilé peut être exécuté par n'importe quel OS, la seule nécessité est de posséder une machine virtuelle Java (JVM) adaptée à cette plateforme.

1.1.4 Linkage

En java, le linkage se fait à l'exécution, cela signifie que si vous modifiez un fichier, vous aurez une seule classe à recompiler, et vous pourrez immédiatement exécuter votre application.

1.2 Le Java procédural

Ce cours vous introduit au Java dans sa dimension procédurale. Le but de ce cours est de vous familiariser avec les instructions de base avant d'aborder les concepts de la programmation objet.

1.2.1 Structure d'une classe

Pour le moment, nous appellerons **classe** un programme Java. Une classe se présente de la sorte :

```
class NomClasse
{
  /*
        Class contents
  */
}
```

Vous remarquez que de façon analogue au C, une classe est un bloc délimité par des accolades et que les commentaires d'écrivent de la même façon. On ecrit les noms des classes en concaténant les mots composant ce nom et en faisant commencer chacun d'eux par une majuscule. N'oubliez pas de ne mettre qu'une seule classe par fichier et de donner au fichier le même nom que celui de cette classe.

1.2.2 Variables

Nous disposons en Java des mêmes types qu'en C. Tous les types mis à votre disposition par Java sont appelé `types primitifs`.

Booléens

L'un deux nous servira toutefois à écrire des choses que l'on rédige habituellement de façon crade en C : `boolean`. Une variable de type `boolean` peut prendre une des deux valeurs `true` et `false`, et seulement une de ces deux valeurs. On déclare et utilise les variables exactement de la même façon qu'en C.

Chaînes de caractères

Il existe un type chaîne de caractères en C. Nous l'examinerons plus détails ultérieurement. Les deux choses à savoir est que le type chaîne de caractères est `String`, et que l'opérateur de concaténation est `+`.

`final`

Les variables dont la déclaration de type est précédée du mot-clé `final` sont non-modifiables. Une fois qu'une valeur leur a été affecté, il n'est plus possible de les modifier. On s'en sert pour représenter des constantes. Les règles typographiqus sont les mêmes : toutes les lettres en majuscules et les mots séparés par des _. Par exemple,

```
final int TAILLE = 100;
```

Déclare une constante `TAILLE` de type `int` initialisée à 100.

1.2.3 Entrées-sorties

La saisie de variables est un calvaire inutile en Java, nous nous en passerons pour le moment. Pour afficher un message, quel que soit son type, on utilise le sous-programme `System.out.print`. Par exemple,

```
System.out.print("Hello World\n");
```

Le sous-programme `System.out.println` ajoute automatiquement un retour à la ligne après l'affichage. On aurait donc pu écrire :

```
System.out.println("Hello World");
```

Il est aussi possible d'intercaler valeurs de variables et chaînes de caractères constantes, on sépare les arguments par des `+`. Par exemple,

```
System.out.println("La valeur de a est " + a +
                   "et celle de b est " + b);
```

1.2.4 Sous-programmes

On définit en Java des sous-programmes de la même façon qu'en C, attention toutefois, les sous-programmes que nous définirons dans ce cours seront déclarés avec le mot-clé `static`. Par exemple, si je veux faire un sous-programme qui retourne le successeur de la valeur passée en argument cela donne

```
class TestJavaProcedural
{

  static int succ(int i)
  {
    return i + 1;
  }

  /*
        Autres sous-programmes
  */
}
```

Si vous oubliez le mot clé `static`, le compilateur vous enverra des insultes. Nous verrons plus tard ce que signifie ce mot et dans quels cas il est possible de s'en passer. Attention : Tous les types primitifs en Java se passent en paramètre par valeur, et cette fois-ci je ne vous mens pas !

1.2.5 Main

Lorsque vous invoquez la machine virtuelle, elle part à la recherche d'un sous-programme particulier appelé `main`. Il est impératif qu'il soit défini comme suit :

```
public static void main(String[] args)
{
  /*
        instructions
  */
}
```

La signification du mot-clé `public` vous sera expliquée ultérieurement. La seule chose dont il faut prendre note est que vous ne devez pas l'oubliez, sinon la machine virtuelle vous enverra des insultes !

1.2.6 Instructions de contrôle de flux

En Java, les instructions `if`, `switch`, `for`, `while` et `do ... while` se comportent de la même façon qu'en C. Donc, pas d'explications complémentaires...

1.2.7 Exemple récapitulatif

Nous avons maintenant tout ce qu'il faut pour écrire un petit programme :

```
public class Exemple
{
    /*
      Retourne le nombre b eleve a la puissance n.
    */
    static int puissance(int b, int n)
    {
```

```
        int res = 1;
        for(int i = 1 ; i <= n ; i++)
            res *= b;
        return res;
    }

    /*
      Affiche {2^k | k = 0, ..., 30}.
    */
    public static void main(String[] args)
    {
        for(int k = 0 ; k <= 30 ; k++)
            System.out.println("2^" + k + " = " + puissance(2, k));
    }
}
```

1.3 Objets

Dans un langage de programmation, un **type** est
- Un ensemble de **valeurs**
- Des **opérations** sur ces valeurs

En plus des types primitifs, il est possible en Java de créer ses propres types. On appelle type construit un type non primitif, c'est-à-dire composé de types primitifs. Certains types construits sont fournis dans les bibliothèques du langage. Si ceux-là ne vosu satisfont pas, vous avez la possibilité de créer vos propres types.

1.3.1 Création d'un type

Nous souhaitons créer un type `Point` dans R^2. Chaque variable de ce type aura deux attributs, une abscisse et une ordonnée. Le type point se compose donc à partir de deux types flottants. Un type construit s'appelle une **classe**. On le définit comme suit :

```
public class Point
{
        float abscisse;
        float ordonnee;
}
```

Les deux attributs d'un objet de type `Point` s'appelle aussi des **champs**. Une fois définie cette classe, le type `Point` est une type comme les autres, il devient donc possible d'écrire

```
Point p, q;
```

Cette instruction déclare deux variables `p` et `q` de type `Point`, ces variables s'appellent des **objets**. Chacun de ces objets a deux attributs auxquels on accède avec la notation pointée. Par exemple, l'abscisse du point `p` est `p.abscisse` et son ordonnée est `p.ordonnee`. Le fonctionnement est, pour le moment, le même que pour les structures en C.

1.3.2 Les méthodes

Non contents d'avoir défini ainsi un ensemble de valeurs, nous souhaiterions définir un ensembe d'opérations sur ces valeurs. Nous allons pour ce faire nous servir de **méthodes**. Une méthode est un sous-programme propre à chaque objet. C'est-à-dire dont le contexte d'exécution est délimité par un objet. Par exemple,

```
public class Point
{
    float abscisse;
    float ordonnee;

    void presenteToi()
    {
        System.out.println("Je suis un point, mes coordonnées sont ("
                           + abscisse + ", " + ordonnee + ")");
    }
}
```

La méthode `presenteToi` s'invoque à partir d'un objet de type `Point`. La syntaxe est `p.presenteToi()` où `p` est de type `Point`. `p` est alors le contexte de l'exécution de `presenteToi` et les champs auquel accèdera cette méthode seront ceux de l'objet `p`. Si par exemple, on ecrit `q.presenteToi()`, c'est `q` qui servira de contexte à l'exécution de `presenteToi`. Lorsque l'on rédige une méthode, l'objet servant de contexte à l'exécution de la méthode est appelé l'**objet courant**.

1.3.3 L'instanciation

Essayons maintenant, impatients et nerveux, de vérifier ce que donnerait l'exécution de

```
public class Point
{
    float abscisse;
    float ordonnee;

    void presenteToi()
    {
        System.out.println("Je suis un point, mes coordonnées sont ("
                           + abscisse + ", " + ordonnee + ")");
    }

    public static void main(String[] args)
    {
        Point p;
        p.abscisse = 1;
        p.ordonnee = 2;
        p.presenteToi();
    }
}
```

Pas de chance, ce programme ne compile pas. Et davantage pointilleux que le C, le compilateur de Java s'arrête au moindre petit souci... Le message d'erreur à la compilation est le suivant :

```
Point.java:15: variable p might not have been initialized
        p.abscisse = 1;
        ^
1 error
```

Que peut bien signifier ce charabia ? Cela signifie que le compilateur a de très sérieuses raisons de penser que `p` contient pas d'objet. En Java, toutes les variables de type construit sont des **pointeurs**. Et les pointeurs non initialisés (avec `malloc` en C) sont des pointeurs ayant la valeur `null`. Pour faire une allocation dynamique en Java, on utilise l'instruction `new`. On fait alors ce que l'on appelle une **instanciation**. La syntaxe est donnée dans l'exemple suivant :

```
p = new Point();
```

Cette instruction crée un objet de type `Point` et place son adresse dans `p`. Ainsi `p` n'est pas un objet, mais juste un pointeur vers un objet de type `Point` qui a été crée par le `new`. Et à partir de ce moment-là, il devient possible d'accéder aux champs de `p`. `new` fonctionne donc de façon similaire à `malloc` et le programme suivant est valide :

```
public class Point
{
    float abscisse;
    float ordonnee;

    void presenteToi()
    {
        System.out.println("Je suis un point, mes coordonnees sont ("
                           + abscisse + ", " + ordonnee + ")");
    }

    public static void main(String[] args)
    {
        Point p;
        p = new Point();
        p.abscisse = 1;
        p.ordonnee = 2;
        p.presenteToi();
    }
}
```

Vous remarquez qu'il n'y a pas de fonction de destruction (`free()` en C). Un programme appelé **Garbage Collector** (ramasse-miette en français) est exécuté dans la machine virtuelle et se charge d'éliminer les objets non référencés.

1.3.4 Les packages

Bon nombre de classes sont prédéfinies en Java, elles sont réparties dans des packages, pour utilisez une classe se trouvant dans un package, on l'importe au début du fichier source. Pour importer un package, on utilise l'instruction `import`.

1.3.5 Le mot-clé `this`

Dans toute méthode, vous disposez d'une référence vers l'objets servant de contexte à l'exécution de la méthode, cette référence s'appelle `this`.

1.4 Tableaux

1.4.1 Déclaration

Un tableau en Java est un objet. Il est nécessaire de le créer par allocation dynamique avec un `new` en précisant ses dimensions. On note

```
T[]
```

le type tableau d'éléments de type `T`. La taille du tableau n'est précisée qu'à l'instanciation. On déclare un tableau `t` d'éléments de type `T` de la façon suivante :

```
T[] t;
```

Par exemple, si l'on souhaite créer un tableau `i` d'éléments de type `int`, on utilise l'instruction :

```
int[] i;
```

1.4.2 Instanciation

Comme un tableau est un objet, il est nécessaire d'instancier pendant l'exécution. On instancie un tableau avec l'instruction

```
new T[taille]
```

Par exemple, si l'on souhaite déclarer et allouer dynamiquement un tableau de 100 entiers, en utilise

```
int[] i = new int[100];
```

1.4.3 Accès aux éléments

On accède aux éléments d'un tableau avec la notation à crochets, par exemple,

```
int[] t = new int[100];
for(int i = 0 ; i < 100 ; i++)
        {
          t[i] = 100 - (i + 1);
          Sytem.out.println(t[i]);
        }
```

On remarque que la variable `i` est déclarée à l'intérieur de la boucle `for`, cette façon de déclarer les variables est très utile en Java. Dans l'exemple,

```
public class ExempleTableau
{
    public static void main(String[] args)
    {
        final int T = 20;
        int[] t = new int[T];
        for (int i = 0 ; i < T ; i++)
            t[i] = i;
        for (int i = 0 ; i < T ; i++)
            System.out.println(t[i]);
    }
}
```

Une tableau est alloué dynamiquement et ses dimensions sont de taille fixée à la compilation.

1.4.4 Longueur d'un tableau

Pour connaître la taille d'un tableau on utilise l'attribut `length`. Par exemple,

```
int[] t = new int[T];
for (int i = 0 ; i < t.length ; i++)
        t[i] = i;
for (int i = 0 ; i < t.length ; i++)
        System.out.println(t[i]);
```

1.4.5 Tableaux à plusieurs dimensions

On crée un tableau à plusieurs dimensions (3 par exemple) en juxtaposant plusieurs crochets. Par exemple,

```
int[][][] t =  new int[2][2][2];
for(int i = 0 ; i < 2 ; i++)
for(int j = 0 ; i < 2 ; j++)
for(int k = 0 ; i < 2 ; k++)
          T=t[i][j][k] = 100*i + 10*i + k;
          Sytem.out.println(t[i][j][k]);
        }
```

On encore,

```
public class ExempleCubique
{
    public static void main(String[] args)
    {
        final int T = 3;
        int[][][] u =  new int[T][T][T];
        for(int i = 0 ; i < T ; i++)
            for(int j = 0 ; j < T ; j++)
                for(int k = 0 ; k < T ; k++)
                    {
                        u[i][j][k] = 100*i + 10*j + k;
                        System.out.println(u[i][j][k]);
                    }

    }
}
```

1.5 Encapsulation

1.5.1 Exemple

Implémentons une file de nombre entiers. Nous avons un exemple dans le fichier suivant :

```
public class FilePublic
{
    /*
      Elements de la file
     */

    int[] entiers;

    /*****************************/

    /*
      Indice de la tete de file et du premier emplacement libre
      dans le tableau.
     */

    int first, firstFree;

    /*****************************/

    /*
      Initialise les attributs de la file.
     */

    public void init(int taille)
    {
        entiers = new int[taille];
        first = firstFree = 0;
    }

    /*****************************/

    /*
      Decale i d'une position vers la droite dans le tableau,
      revient au debut si i deborde.
     */

    public int incrementeIndice(int i)
    {
        i++;
        if (i == entiers.length)
            i = 0;
        return i;
    }

    /*****************************/

    /*
      Retourne vrai si et seulement si la file
      est pleine.
     */
```

```
public boolean estPlein()
{
    return first == incrementeIndice(firstFree);
}

/****************************/

/*
  Retourne vrai si et seulement si
  la file est vide.
 */

public boolean estVide()
{
    return first == firstFree;
}

/****************************/

/*
  Ajoute l'element n dans la file.
 */

public void enfile(int n)
{
    if (!estPlein())
        {
            entiers[firstFree] = n;
            firstFree = incrementeIndice(firstFree);
        }
}

/****************************/

/*
  Supprime la tete de file.
 */

public void defile()
{
    if (!estVide())
        first = incrementeIndice(first);
}

/****************************/

/*
  Retourne la tete de file.
 */

public int premier()
{
    if (! estVide())
```

```
            return entiers[first];
        return 0;
    }
}
```

Un exemple typique d'utilisation de cette file est donné ci-dessous :

```
public class TestFilePublic
{
    public static void main(String[] args)
    {
        FilePublic f = new FilePublic();
        f.init(20);
        for (int i = 0 ; i < 30 ; i+=2)
            {
                f.enfile(i);
                f.enfile(i+1);
                System.out.println(f.premier());
                f.defile();
            }
        while(!f.estVide())
            {
                System.out.println(f.premier());
                f.defile();
            }
    }
}
```

Si vous travaillez en équipe, et que vous êtes l'auteur d'une classe `FilePublic`, vous n'aimeriez pas que vos collègues programmeurs s'en servent n'importe comment ! Vous n'aimeriez pas par exemple qu'ils manipulent le tableau `entiers` ou l'attribut `first` sans passer par les méthodes, par exemple :

```
FilePublic f = new FilePublic();
f.init(20);
f.entiers[4] = 3;
f.first += 5;
/* ... arretons la les horreurs ... */
```

Il serait appréciable que nous puissions interdire de telles instructions. C'est-à-dire forcer l'utilisateur de la classe à passer par les méthodes pour manier la file.

1.5.2 Visibilité

La **visibilité** d'un identificateur (attribut, méthode ou classe) est l'ensemble des endroits dans le code ou il est possible de l'utiliser. Si un identificateur et précédé du mot clé `public`, cela signifie qu'il est visible partout. Si un identificateur et précédé du mot clé `private`, cela signifie qu'il n'est visible qu'à l'intérieur de la classe. Seule l'instance à qui cet identificateur appartient pourra l'utiliser. Par exemple :

```
public class FilePrivate
{
    private int[] entiers;
    private int first, firstFree;

    public void init(int taille)
    {
        entiers = new int[taille];
        first = firstFree = 0;
    }
```

```
    private int incrementeIndice(int i)
    {
        i++;
        if (i == entiers.length)
            i = 0;
        return i;
    }

    public boolean estPlein()
    {
        return first == firstFree + 1;
    }

    public boolean estVide()
    {
        return first == incrementeIndice(firstFree);
    }

    public void enfile(int n)
    {
        if (!estPlein())
            {
                entiers[firstFree] = n;
                firstFree = incrementeIndice(firstFree);
            }
    }

    public void defile()
    {
        if (!estVide())
            first = incrementeIndice(first);
    }

    public int premier()
    {
        if (! estVide())
            return entiers[first];
        return 0;
    }
}
```

On teste cette classe de la même façon :

```
public class TestFilePrivate
{
    public static void main(String[] args)
    {
        FilePrivate f = new FilePrivate();
        f.init(20);
        for (int i = 0 ; i < 30 ; i+=2)
            {
                f.enfile(i);
                f.enfile(i+1);
                System.out.println(f.premier());
                f.defile();
```

```
            }
        while(!f.estVide())
            {
                System.out.println(f.premier());
                f.defile();
            }
    }
}
```

S'il vous vient l'idée saugrenue d'exécuter les instructions :

```
FilePrivate f = new FilePrivate();
f.init(20);
f.entiers[4] = 3;
f.first += 5;
/* ... arretons la les horreurs ... */
```

vous ne passerez pas la compilation. Comme les champs `entiers` et `first` sont `private`, il est impossible de les utiliser avec la notation pointée. Cela signifie qu'ils ne sont accessibles que depuis l'instance de la classe `FilePrivate` qui sert de contexte à leur exécution. De cette façon vous êtes certain que votre classe fonctionnera correctement. En déclarant des champs privés, vous avez caché les divers détails de l'implémentation, cela s'appelle l'**encapsulation**. Celle-ci a pour but de faciliter le travail de tout programmeur qui utilisera cette classe en masquant la complexité de votre code. Les informations à communiquer à l'utilisateur de la classe sont la liste des méthodes publiques. A savoir

```
public class FilePrivate
{
    public void init(int taille){/*...*/}
    public boolean estPlein(){/*...*/}
    public boolean estVide(){/*...*/}
    public void enfile(int n){/*...*/}
    public void defile(){/*...*/}
    public int premier(){/*...*/}
}
```

On remarque non seulement qu'il plus aisé de comprendre comment utiliser la file en regardant ces quelques méthodes mais surtout que la façon dont a été implémenté la file est totalement masquée.

1.5.3 Constructeur

Supposons que notre utilisateur oublie d'invoquer la méthode `init(int taille)`, que va-t-il se passer ? Votre classe va planter. Comment faire pour être certain que toutes les variables seront initialisées ? Un **constructeur** est un sous-programme appelé automatiquement à la création de tout objet. Il porte le même nom que la classe et n'a pas de valeur de retour. De plus, il est possible de lui passer des paramètres au moment de l'instanciation. Remplaçons `init` par un constructeur :

```
public class FileConstructeur
{
    private int[] entiers;
    private int first, firstFree;

    FileConstructeur(int taille)
    {
        entiers = new int[taille];
        first = firstFree = 0;
    }

    private int incrementeIndice(int i)
```

```
    {
        i++;
        if (i == entiers.length)
            i = 0;
        return i;
    }

    public boolean estPlein()
    {
        return first == firstFree + 1;
    }

    public boolean estVide()
    {
        return first == incrementeIndice(firstFree);
    }

    public void enfile(int n)
    {
        if (!estPlein())
            {
                entiers[firstFree] = n;
                firstFree = incrementeIndice(firstFree);
            }
    }

    public void defile()
    {
        if (!estVide())
            first = incrementeIndice(first);
    }

    public int premier()
    {
        if (! estVide())
            return entiers[first];
        return 0;
    }
}
```

On peut alors l'utiliser sans la méthode `init`,

```
public class TestFileConstructeur
{
    public static void main(String[] args)
    {
        FileConstructeur f = new FileConstructeur(20);
        for (int i = 0 ; i < 30 ; i+=2)
            {
                f.enfile(i);
                f.enfile(i+1);
                System.out.println(f.premier());
                f.defile();
            }
        while(!f.estVide())
            {
```

```
                System.out.println(f.premier());
                f.defile();
            }
        }
    }
```

1.5.4 Accesseurs

Il est de bonne programmation de déclarer tous les attributs en privé, et de permettre leur accès en forçant l'utilisateur à passer par des méthodes. Si un attribut X est privé, on crée deux méthodes $getX$ et $setX$ permettant de manier X. Par exemple,

```
public class ExempleAccesseurs
{
    private int foo;

    public int getFoo()
    {
        return foo;
    }

    public void setFoo(int value)
    {
        foo = value;
    }
}
```

On ainsi la possibilité de manier des attributs privés indirectement.

1.5.5 Surcharge

Il est possible de definir dans une même classe plusieurs fonctions portant le même nom. Par exemple,

```
public class FileSurcharge
{
    private int[] entiers;
    private int first, firstFree;

    FileSurcharge(int taille)
    {
        entiers = new int[taille];
        first = firstFree = 0;
    }

    FileSurcharge(FileSurcharge other)
    {
        this(other.entiers.length);
        for(int i = 0 ; i < other.entiers.length ; i++)
            entiers[i] = other.entiers[i];
        first = other.first;
        firstFree = other.firstFree;
    }

    private int incrementeIndice(int i)
    {
        i++;
```

```
        if (i == entiers.length)
            i = 0;
        return i;
    }

    public boolean estPlein()
    {
        return first == firstFree + 1;
    }

    public boolean estVide()
    {
        return first == incrementeIndice(firstFree);
    }

    public void enfile(int n)
    {
        if (!estPlein())
            {
                entiers[firstFree] = n;
                firstFree = incrementeIndice(firstFree);
            }
    }

    public void defile()
    {
        if (!estVide())
            first = incrementeIndice(first);
    }

    public int premier()
    {
        if (! estVide())
            return entiers[first];
        return 0;
    }
}
```

On remarque qu'il y a deux constructeurs. L'un prend la taille du tableau en paramètre et l'autre est un constructeur de copie. Selon le type de paramètre passé au moment de l'instanciation, le constructeur correspondant est exécuté. Dans l'exemple ci-dessous, on crée une file g en applicant le constructeur de copie à la file f.

```
public class TestFileSurcharge
{
    public static void main(String[] args)
    {
        FileSurcharge f = new FileSurcharge(20);
        for (int i = 0 ; i < 30 ; i+=2)
            {
                f.enfile(i);
                f.enfile(i+1);
                System.out.println(f.premier());
                f.defile();
            }
        FileSurcharge g = new FileSurcharge(f);
```

```
        while(!f.estVide())
            {
                System.out.println(f.premier());
                f.defile();
            }
        while(!g.estVide())
            {
                System.out.println(g.premier());
                g.defile();
            }
    }
}
```

1.6 Héritage

1.6.1 Héritage

Le principe

Quand on dispose d'une classe c, on a la possibilité de l'agrandir (ou de l'étendre) en créant une deuxième classe c'. On dit dans ce cas que c' **hérite de** c, ou encore que c est la **classe mère** et c' la **classe fille**. Par exemple, considérons les deux classes suivantes :

```
public class ClasseMere
{
    private final int x;

    public ClasseMere(int x)
    {
        this.x = x;
    }

    public int getX()
    {
        return x;
    }
}

public class ClasseFille extends ClasseMere
{
    private final int y;

    public ClasseFille(int x, int y)
    {
        super(x);
        this.y = y;
    }

    public int getY()
    {
        return y;
    }
}
```

Le mot-clé `extends` signifie **hérite de**. Le mot-clé `super` est le constructeur de la classe mère. Tout ce qui peut être fait avec la classe mère peut aussi être fait avec la classe fille. Notez donc que la classe fille est une **extension** de la classe mère. On peut par exemple utiliser la classe fille de la façon suivante :

```
public class TestClasseFille
{
    public static void main(String[] args)
    {
        ClasseFille o = new ClasseFille(1, 2);
        System.out.println("(" + o.getX() + ", " +
                           o.getY() + ")");
    }
}
```

Notes bien que comme la classe mère possède une méthode `getX`, la classe fille la possède aussi. Et l'attribut `x` de tout instance de la classe mère est aussi un attribut de toute instance de la classe fille.

Héritage simple Vs. héritage multiple

En java, une classe ne peut avoir qu'une seule classe mère. Dans d'autres langages (comme le `C++`) cela est permis, mais pour des raisons de fiabilité, Java l'interdit.

`Object`

En java, toutes les classes héritent implicitement d'une classe `Object`.

1.6.2 Polymorphisme

Considérons l'exemple suivant :

```
public class TestClasseFillePolymorphisme
{
    public static void main(String[] args)
    {
        ClasseMere o = new ClasseFille(1, 2);
        System.out.println("(" + o.getX() + ", " +
                           ((ClasseFille)o).getY() + ")");
    }
}
```

On remarque d'une part que *o* référence un objet de la classe fille de son type. Cela s'appelle le **polymorphisme**. D'autre part, si l'on souhaite effectuer des opérations spécifiques aux objets de la classe fille, il est nécessaire d'effectuer un cast.

1.6.3 Redéfinition de méthodes

La méthode `toString()` appartient initialement à la classe `Object`. Elle est donc héritée par toutes les classes. Vous avez la possibilité de la redéfinir, c'est à dire de remplacer la méthode par défaut par une méthode davantage adaptée à la classe en question.

1.6.4 Interfaces

Une interface est un ensemble de constantes et de méthodes vides. Il est impossible d'instancier une interface. Il est nécessaire que la classe qui hérite de l'interface (mot-clé `implements`) définisse les méthodes qu'elle contient. En voici un exemple d'utilisation :

```
interface Presentable
{
    public void presenteToi();
}

public class ExempleInterface implements Presentable
{
    public int x, y;

    public void presenteToi()
    {
        System.out.println("Je suis un ExempleInterface, x = "
                           + x + " et y = " + y + ".");
    }

    public static void main(String[] args)
    {
        ExempleInterface t = new ExempleInterface();
```

```
        t.x = 1;
        t.y = 2;
        Presentable j = t;
        j.presenteToi();
    }
}
```

Vous remarquez que la méthode `PresenteToi` est implémentée dans la classe fille (sinon ça ne compilerait pas). Comme `t` est aussi de type `Presentable` (en vertu du polymorphisme), il est possible de le placer dans `j`. Comme tout objet `Presentable` possède une méthode `PresenteToi`, on peut invoquer cette méthode depuis `j`.

1.6.5 Classes Abstraites

Nous voulons représenter des sommes dans des devises différentes et implémenter automatiquement les conversions. Nous allons créer une classe devise qui contiendra l'attribut somme et nous utiliserons l'héritage pour implémenter les spécificités des diverses devises (Euros, Dollars, Livres, ...).

```
public abstract class Devise
{
    private double somme = 0;

    public abstract double getCours();

    public void setSomme(Devise d)
    {
        this.somme = d.somme*d.getCours()/this.getCours();
    }

    public void setSomme(double somme)
    {
        this.somme = somme;
    }

    public String toString()
    {
        return "somme = " + somme + " ";
    }
}
```

La classe devise contient une méthode `setSomme` qui est surchargée et qui peut prendre en paramètre soit une somme exprimée dans la bonne unité, soit une autre devise.

```
public class Euros extends Devise
{
    public Euros(Devise d)
    {
        setSomme(d);
    }

    public Euros(double somme)
    {
        setSomme(somme);
    }

    public double getCours()
    {
```

```
        return 1.4625;
    }

    public String toString()
    {
        return super.toString() + " Euros";
    }
}
```

On remarque qu'il est impossible d'implémenter `getCours()` car le cours varie selon la devise. La classe `Euros` hérite de la classe `Devise` et implémente `getCours()`. La classe `Dollars` fonctionne de façon similaire :

```
public class Dollars extends Devise
{
    public Dollars(Devise d)
    {
        setSomme(d);
    }

    public Dollars(double somme)
    {
        setSomme(somme);
    }

    public double getCours()
    {
        return 1.;
    }

    public String toString()
    {
        return super.toString() + " Dollars";
    }
}
```

Ces classes permettent de prendre en charge automatiquement les conversions entre devises :

```
public class TestDevises
{
    public static void main(String[] args)
    {
        Dollars d = new Dollars(12);
        System.out.println(d);
        Euros e = new Euros(d);
        System.out.println(e);
        Livres l = new Livres(e);
        System.out.println(l);
    }
}
```

1.7 Exceptions

Le mécanisme des exceptions en Java (et dans d'autres langages objets) permet de gérer de façon élégante les erreurs pouvant survenir à l'exécution. Elles présentent trois avantages :

- Obliger les programmeurs à gérer les erreurs.
- Séparer le code de traitement des erreurs du reste du code.
- Rattraper les erreurs en cours d'exécution.

1.7.1 Rattraper une exception

Lorsqu'une séquence d'instructions est susceptible d'occasionner une erreur, on la place dans un bloc `try`. Comme ci-dessous :

```
try
{
  /*
        bloc d'instructions
  */
}
```

On place juste après ce bloc un bloc `catch` qui permet de traiter l'erreur :

```
try
{
  /*
        bloc d'instructions
  */
}
catch(nomErreur e)
{
  /*
        Traitement de l'erreur
  */
}
```

Une **exception** est une **erreur pouvant être rattrapée en cours d'exécution**. Cela signifie que si une erreur survient, une exception est **levée**, le code du `try` est interrompu et le code contenu dans le `catch` est exécuté. Par exemple,

```
try
{
  int i = 0;
  while(true)
  {
    t[i++]++;
  }
}
catch(ArrayIndexOutOfBoundException e)
{
  System.out.println("An exception has been raised.");
}
```

Dans le code précédent l'indice `i` est incrémenté jusqu'à ce que cet indice dépasse la taille du tableau. Dans ce cas l'exception `ArrayIndexOutOfBoundException` est levée et le code correspondant à cette exception dans le `catch` est exécuté.

1.7.2 Méthodes levant des exceptions

Si vous rédigez une méthode susceptible de lever une exception, vous **devez** le déclarer en ajoutant à son entête l'expression `throws <listeExceptions>`. Par exemple,

```
public void bourreTableau() throw ArrayIndexOutOfBoundException
{
  int i = 0;
  while(true)
  {
    t[i++]++;
  }
}
```

Vous remarquez qu'il n'y a pas de `try ... catch`. Cela signifie que l'erreur doit être rattrapée dans le sous-programme appelant. Par exemple,

```
try
{
  bourreTableau();
}
catch(ArrayIndexOutOfBoundException e)
{
  System.out.println("Il fallait s'y attendre...");
}
```

Bref, lorsqu'une instruction est susceptible de lever une exception, vous devez soit la rattraper tout de suite, soit indiquer dans l'entête du sous-programme qu'une exception peut se propager et qu'elle doit donc être traitée dans le sous-programme appelant.

1.7.3 Propagation d'une exception

Par traiter une exception, on entend soit la rattraper tout de suite, soit la **propager**. Par propager, on entend laisser le sous-programme appelant la traiter. Il est possible lorsque l'on invoque un sous-programme levant une exception, de la propager. Par exemple,

```
public void appelleBourreTableau() throws ArrayIndexOutOfBoundException
{
  bourreTableau();
}
```

On observe que `appelleBourreTableau` se contente de transmettre l'exception au sous-programme appelant. Ce mode de fonctionnement fait qu'une exception non rattrapée va se propager dans la pile d'appels de sous-programmes jusqu'à ce qu'elle soit rattrapée. Et si elle n'est jamais rattrapée, c'est la JVM, qui va le faire.

1.7.4 Définir une exception

Une exception est tout simplement une classe héritant de la classe `Exception`.

```
class MonException extends Exception
{
    public MonException()
    {
        System.out.println("Exception monException has been raised...");
    }

    public String toString()
    {
```

```
        return "You tried to do an illegal assignement !";
    }
}
```

1.7.5 Lever une exception

On lève une exception en utilisant la syntaxe `throw new <nomexception>(<parametres>)`. L'instanciation de l'exception se fait donc en même temps que sa levée. Par exemple,

```
try
{
  /*
  ......
  */
  throw new MonException();
  /*
  ......
  */
}
catch(MonException e)
{
  System.out.println(e);
}
```

1.7.6 Rattraper plusieurs exceptions

Il se peut qu'une même instruction (ou suite d'instruction) soit susceptible de lever des exceptions différentes, vous pouvez dans ce cas placer plusieurs `catch` à la suite du même `try`.

```
try
{
  bourreTableau();
}
catch(ArrayIndexOutOfBoundException e)
{
  System.out.println("Il fallait s'y attendre...");
}
catch(Exception e)
{
  System.out.println(e);
}
```

Notez bien que comme toute exception hérite de `Exception`, le deuxième `catch` sera exécuté quelle que soit l'exception levée (sauf si bien entendu le premier `catch` est exécuté).

1.7.7 Finally

Il est quelquefois nécessaire qu'une section de code s'exécute quoi qu'il advienne (pour par exemple fermer un fichier). On place dans ce cas une section `finally`.

```
try
{
  /*
  ......;
  */
}
```

```
catch(/* ...*/)
{
  /*
  .....
  */
}
finally
{
  /*
  .....
  */
}
```

Par exemple,

```
class MonException extends Exception
{
    public String toString()
    {
        return "Fallait pas invoquer cette methode...";
    }
}

public class Finally
{
    public static void main(String[] args)
    {
        Finally f  = new Finally();
        try
            {
                try
                    {
                        throw new MonException();
                    }
                catch(MonException e)
                    {
                        throw new MonException();
                    }
                finally
                    {
                        System.out.println("Tu t'afficheras quoi qu'il advienne !");
                    }
            }
        catch(Exception e)
            {
                System.out.println(e);
            }
    }
}
```

1.7.8 RuntimeException

Je vous vous ai menti au sujet de `ArrayIndexOutOfBoundException`, il s'agit d'une exception héritant de `RunTimeException` qui lui-même hérite de `Exception`. Un code Java dans lequel une `RunTimeException` n'est pas traitée passera la compilation. Par contre, `MonException` n'est pas une `RunTimeException` et doit être rattrapée !

1.8 Interfaces graphiques

1.8.1 Fenêtres

La classe `JFrame`

Une fenêtre est un objet de type `JFrame`, classe à laquelle on accède avec la commande d'importation `import javax.swing.*`. Nous utiliserons les méthodes suivantes :

- `public void setTitle(String title)`, pour définir le titre de la fenêtre.
- `public void setVisible(boolean b)`, pour afficher la fenêtre.
- `public void setSize(int width, height)`, pour préciser les dimensions de la fenêtre.
- `public void setDefaultCloseOperation(int operation)`, pour déterminer le comportement de la fenêtre lors de sa fermeture.

Exemples

Voici un code de création d'une fenêtre vide.

```
import javax.swing.*;

public class PremiereFenetre
{
    public static void main(String[] args)
    {
        JFrame f = new JFrame();
        f.setVisible(true);
        f.setTitle("My first window !");
        f.setSize(200, 200);
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

La façon suivante nous sera utile pour gérer les événements :

```
import javax.swing.*;

public class DeuxiemeFenetre extends JFrame
{
    public DeuxiemeFenetre()
    {
        super();
        setTitle("My second window !");
        setSize(200, 200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
    }

    public void affiche()
    {
        setVisible(true);
    }

    public static void main(String[] args)
    {
        DeuxiemeFenetre f = new DeuxiemeFenetre();
        f.affiche();
    }
}
```

1.8.2 Un premier objet graphique

Nous allons ajouter un bouton dans notre fenêtre. Les boutons sont de type `JButton`, le constructeur de cette classe prend en paramètre le libellé du bouton. Un des attributs de la classe JFrame est un objet contenant tous les composants graphiques de la fenêtre, on accède à cet objet par la méthode `public Container getContentPane()`. Tout `Container` possède une méthode `public Component add(Component comp)` permettant d'ajouter n'importe quel composant graphique (par exemple un `JButton`...). On ajoute donc un Bouton de la façon suivante `getContentPane().add(new JButton("my First JButton"));`.

```
import javax.swing.*;
import java.awt.*;

public class PremiersJButtons extends JFrame
{
    public PremiersJButtons()
    {
        super();
        setTitle("My third window !");
        setSize(200, 200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        getContentPane().setLayout(new FlowLayout());
        getContentPane().add(new JButton("my First JButton"));
        getContentPane().add(new JButton("my Second JButton"));
        getContentPane().add(new JButton("my Third JButton"));
        setVisible(true);
    }

    public static void main(String[] args)
    {
        new PremiersJButtons();
    }
}
```

L'instruction `getContentPane().setLayout(new FlowLayout());` sert à préciser de quelle façon vous souhaitez que les composants soient disposés dans la fenêtre. Nous reviendrons dessus.

1.8.3 Ecouteurs d'événements

Un écouteur d'événement est un objet associé à un composant graphique. Cet objet doit implémenter l'interface `ActionListener` et de ce fait contenir une méthode `public void actionPerformed(ActionEvent e)`. Cette méthode est appelée à chaque fois qu'un événement se produit sur le composant. L'objet `ActionEvent` contient des informations sur l'événement en question.

1.8.4 Premier exemple

Dans ce premier exemple, nous allons définir une classe `public class PremierEcouteur extends JFrame implements ActionListener`. Elle servira donc à la fois de fenêtre et d'écouteur d'événements. L'objet de type `ActionEvent` passé en paramètre contient une méthode `public Object getSource()` retournant l'objet ayant déclenché l'événement.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class PremierEcouteur extends JFrame implements ActionListener
{
    JButton[] jButtons;
```

```
    public void actionPerformed(ActionEvent e)
    {
        int k = 0;
        while(jButtons[k++] != e.getSource());
        System.out.println("click on JButton " + k);
    }

    public PremierEcouteur()
    {
        super();
        jButtons = new JButton[3];
        setTitle("My fourth window !");
        setSize(200, 200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        getContentPane().setLayout(new FlowLayout());
        jButtons[0] = new JButton("my First JButton");
        jButtons[1] = new JButton("my Second JButton");
        jButtons[2] = new JButton("my Third JButton");
        for (int i = 0 ; i < 3 ; i++)
            {
                getContentPane().add(jButtons[i]);
                jButtons[i].addActionListener(this);
            }
        setVisible(true);
    }

    public static void main(String[] args)
    {
        new PremierEcouteur();
    }
}
```

1.8.5 Classes anonymes

Les classes anonymes sont des classes que l'on peut créer "à la volée", c'est-à-dire au moment de leur instanciation.. On s'en sert généralement pour implémenter une interface, en l'occurrence `ActionListener`. Voici une autre implémentation faisant intervenir des classes anonymes.

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class EcouteurAnonyme extends JFrame
{
    public EcouteurAnonyme()
    {
        super();
        JButton[] jButtons = new JButton[3];
        setTitle("My fourth window !");
        setSize(200, 200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        getContentPane().setLayout(new FlowLayout());
        jButtons[0] = new JButton("my First JButton");
        jButtons[1] = new JButton("my Second JButton");
```

```
        jButtons[2] = new JButton("my Third JButton");
        for (int i = 0 ; i < 3 ; i++)
            getContentPane().add(jButtons[i]);
        jButtons[0].addActionListener(new ActionListener()
                {
                    public void actionPerformed(ActionEvent e)
                    {
                        System.out.println("click on First JButton");
                    }
                }
                );
        jButtons[1].addActionListener(new ActionListener()
                {
                    public void actionPerformed(ActionEvent e)
                    {
                        System.out.println("click on Second JButton");
                    }
                }
                );
        jButtons[2].addActionListener(new ActionListener()
                {
                    public void actionPerformed(ActionEvent e)
                    {
                        System.out.println("click on Third JButton");
                    }
                }
                );

        setVisible(true);
    }

    public static void main(String[] args)
    {
        new EcouteurAnonyme();
    }
}
```

1.8.6 Gestionnaires de mise en forme

Un gestionnaire de mise en forme (Layout manager) est un objet permettant de disposer les composants et les conteneurs dans la fenêtre. Nous avons utilisé un gestionnaire de type `FlowLayout`, qui dispose les composants les uns à coté des autres (ou au dessus des autres si ce n'est pas possible) dans leur ordre d'ajout. Le gestionnaire `GridLayout` représente le conteneur comme une grille

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class TestGridLayout extends JFrame implements ActionListener
{
    JButton[] jButtons;

    public void actionPerformed(ActionEvent e)
    {
        int k = 0;
```

```
        while(jButtons[k++] != e.getSource());
        System.out.println("click on JButton " + k);
    }

    public TestGridLayout()
    {
        super();
        jButtons = new JButton[4];
        setTitle("One More Window !");
        setSize(200, 200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        getContentPane().setLayout(new GridLayout(2, 2));
        jButtons[0] = new JButton("my First JButton");
        jButtons[1] = new JButton("my Second JButton");
        jButtons[2] = new JButton("my Third JButton");
        jButtons[3] = new JButton("my Fourth JButton");
        for (int i = 0 ; i < 4 ; i++)
            {
                getContentPane().add(jButtons[i]);
                jButtons[i].addActionListener(this);
            }
        setVisible(true);
    }

    public static void main(String[] args)
    {
        new TestGridLayout();
    }
}
```

1.8.7 Un exemple complet : Calcul d'un carré

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class Carre extends JFrame implements KeyListener
{
    JTextField operandTextField;
    JLabel resultTextField;

    public void keyPressed(KeyEvent e){}

    public void keyReleased(KeyEvent e)
    {
        try
        {
        int k = Integer.parseInt(operandTextField.getText());
        k *= k;
        resultTextField.setText(Integer.toString(k));
        }
        catch(Exception ex)
        {
        if(resultTextField != null)
```

```
                resultTextField.setText("");
            }
        }

        public void keyTyped(KeyEvent e){}

        public Carre()
        {
            super();
            setSize(200, 200);
            setTitle("Square computer !");
            setDefaultCloseOperation(EXIT_ON_CLOSE);
            getContentPane().setLayout(new GridLayout(2, 2));
            getContentPane().add(new JLabel("x = "));
            operandTextField = new JTextField();
            operandTextField.addKeyListener(this);
            getContentPane().add(operandTextField);
            getContentPane().add(new JLabel("x^2 = "));
            resultTextField = new JLabel();
            getContentPane().add(resultTextField);
            setVisible(true);
        }

        public static void main(String[] args)
        {
            new Carre();
        }
}
```

1.9 Collections

1.9.1 Packages

L'instruction `import` que vous utilisez depuis le début sert à localiser une classe parmi les packages standard de Java. Si par exemple vous importez `import java.awt.event.*`, cela signifie que vous allez utiliser les classes se trouvant dans le package qui se trouve dans le package `event` qui se trouve dans le package `awt` qui se trouve dans le package `java` (qui est la bibliothèque standard).

Créer ses propres packages

Il possible pour un programmeur de créer ses propres packages, deux choses sont nécessaires :
- Utilisez la même arborescence dans le système de fichier que celle des packages
- Ajouter au début de chaque source sa position dans l'arborescence.

Par exemple,

```
package collections;

public class ExemplePackage
{
    public static void presenteToi()
    {
        System.out.println("Je suis collections.ExemplePackage");
    }

    public static void main(String[] args)
    {
        presenteToi();
    }
}
```

On exécute et compile ce fichier en se plaçant non pas dans le répertoire où se trouve les source, mais dans le répertoire ou se trouve le package. Par conséquent, l'exécution en ligne de commande se fait avec `javac collections.ExemplePackage`

Utiliser ses propres packages

Vos packages s'utilisent comme les packages de la bibliothèque standard. Si le `main` se trouve en dehors de votre package, la variable d'environnement `CLASSPATH` sera utilisée pour localiser votre package.

Visibilité

En l'absence de mots-clés `public` (tout le monde), `private` (seulement la classe) et `protected` (classe + classes dérivées), une visibilité par défaut est appliquée. Par défaut, la visibilité des éléments est limitée au package.

1.9.2 Types paramétrés

L'utilisation du type `Object` pour faire de la généricité en Java se paye avec des instruction de ce type

```
maCollection.add(new ClasseFille());
ClasseFille f = (ClasseFille)maCollection.get(0);
```

Il est nécessaire d'effectuer un cast particulièrement laid pour passer la compilation. On prend par conséquent le risque qu'une exception soit levée à l'exécution s'il s'avère que l'objet récupéré n'est pas de type `ClassFille`. Les types paramétrés fournissent un moyen élégant de résoudre ce problème.

Java Vs C++

Le mécanisme des classes paramétrées en Java en bien plus fin que les templates utilisées en C++. Les templates ne sont que de vulgaires rechercher/remplacer, alors qu'en Java, le code compilé est le même que si les types ne sont pas paramétrés. Les types paramétrés de java ne sont donc qu'un moyen pour le compilateur de s'assurer que vous faites les choses proprement en vous obligeant à déclarer les types que vous placez dans les collections.

La syntaxe

La syntaxe est par contre la même qu'en C++ : il suffit d'utiliser des chevrons pour placer le paramètre. Par exemple,

```
public class ExempleClasseParametree<T>
{
    T data;

    public ExempleClasseParametree(T data)
        {
            this.data = data;
        }

    public T get()
    {
        return data;
    }
}
```

On est amené naturellement à se demander si `ExempleClasseParametree<String>` hérite de `ExempleClasseParametre`
Un exercice traite cette question.

Contraintes sur le paramètre

Il est souvent nécessaire que le type servant de paramètre vérifie certaines propriétés. Si vous souhaitez par exemple que l'objet hérite de `Comparable`, vous avez la possibilité de poser cette contrainte avec le paramètre `<T extends Comparable>`. La définition ci-avant est incomplète, le soin vous est laissé de la compléter...

1.9.3 Collections standard

La distribution officielle de Java est fournie avec plusieurs milliers de classes. Les `Collections` sont des structures de données permettant d'optimiser des opérations comme le tri, la recherche de plus petit élément, etc. Les collections sont donc des regroupements d'objets.

Classification des collections

L'interface `Collection<T>` décrit ce que doit faire chaque collection. Elle plusieurs sous-interfaces, parmi lesquelles on trouve :

- `List<T>` conserve l'ordre des éléments dans l'ordre dans lequel le programmeur souhaite qu'ils soient.
- `Set<T>` ne tient pas compte de l'ordre des éléments.
- `Queue<T>` est optimisé pour récupérer rapidement le plus petit élément.

Ces interfaces sont implémentées dans diverses classes ou divisées avec des sous-interfaces permettent d'obtenir des fonctionnalités plus précises.

Parcours d'une collection

Il existe, de façon analogue à que l'on observe en C++, une syntaxe permettant simplement de parcourir une collection qui est :

```
for (T x : c)
{
  /* ... */
}
```

Dans la spécification ci dessus, `c` est la collection, `T` le type (ou un sur-type) des objets de la collection et `x` une variable (muette) dans laquelle sera placé tour à tour chaque objet de la collection. Il n'est possible d'utiliser cette syntaxe que si le type de `c` implémente `Iterable<T>`.

Détails

Parmi les classes implémentant `List<T>`, on trouve `ArrayList<T>`, `Vector<T>` et `LinkedList<T>`. Les deux premières sont des tableaux et la dernière une liste chaînée. L'interface `Set<T>` possède une sous-classe `HashSet<T>` et une sous-interface `SortedSet<T>`. Dans un `SortedSet<T>`, les éléments sont automatiquement maintenus dans l'ordre. Parmi les classes implémentant `SortedSet<T>` se trouve `TreeSet<T>` (arbre binaire de recherche).

1.10 Threads

1.10.1 Le principe

Un programme peut être découpé en processus légers, ou **threads**, s'exécutant chacun de leur coté. On définit un thread de deux façon :

- En héritant de la classe `Thread`
- En implémentant l'interface `Runnable`

Bien que la première solution soit généralement plus commode, la deuxième est quelquefois le seul moyen d'éviter l'héritage multiple. Nous détaillerons le premier cas, le deuxième est décrit dans la documentation.

La classe `Thread`

La classe `Thread` dispose entre autres de deux méthodes

- `public void start()` qui est la méthode permettant de démarrer l'exécution du thread.
- `public void run()` qui est la méthode automatiquement invoquée par start quand le thread est demarré.

Voici un exemple d'utilisation :

```
public class BinaireAleatoire extends Thread
{
    private int value;
    private int nbIterations;

    public BinaireAleatoire(int value, int nbIterations)
    {
        this.value = value;
        this.nbIterations = nbIterations;
    }

    public void run()
    {
        for (int i = 1 ; i <= nbIterations ; i++)
            System.out.print(value);
    }

    public static void main(String[] args)
    {
        Thread un = new BinaireAleatoire(1, 30000);
        Thread zero = new BinaireAleatoire(0, 30000);
        un.start();
        zero.start();
    }
}
```

L'interface `Runnable`

Le constructeur de la classe `Thread` est surchargé pour prendre un paramètre une instance `Runnable`. `Runnable` est une interface contenant une méthode `public void run()`, celle-ci sera invoquée par le thread au moment de son lancement.

1.10.2 Synchronisation

Une section critique se construit avec le mot-clé `synchronized`.

Méthodes synchronisées

Une méthode synchronisée ne peut être exécutée que par un thread à la fois. On synchronise une méthode en plaçant le mot clé `synchronized` dans sa définition.

Instructions synchronisées

On synchronise des instructions en les plaçant dans un bloc `synchronized(o){...}`, où `o` est l'objet ne pouvant être accédé par deux threads simultanément.

1.10.3 Mise en Attente

Un thread peut décider de se mettre en attente s'il a besoin pour s'exécuter de données qui ne sont pas encore disponibles. On gère cela avec les instructions suivantes :

- `public void wait() throws InterruptedException` met le thread en attente.
- `public void notify()` réveille un thread en attente
- `public void notifyAll()` réveille tous les thread en attente

On place en général ces instructions dans une section critique. Un `wait()` libère le verrou pour autoriser d'autres threads à accéder à la ressource. Un `notify()` choisit un des objets placés en attente sur la même ressource, lui rend le verrou, et relance son exécution. Par exemple,

```
public class Counter
{
    private int value = 0;
    private int upperBound;
    private int lowerBound;

    public Counter(int lowerBound, int upperBound)
    {
        this.upperBound = upperBound;
        this.lowerBound = lowerBound;
    }

    public synchronized void increaseCounter() throws InterruptedException
    {
        if (value == upperBound)
            wait();
        else
            {
                value++;
                System.out.println("+ 1 = " + value);
                Thread.sleep(50);
                if (value == lowerBound + 1)
                    notify();
            }
    }

    public synchronized void decreaseCounter() throws InterruptedException
    {
        if (value == lowerBound)
            wait();
        else
            {
                value--;
                System.out.println("- 1 = " + value);
                Thread.sleep(50);
```

```
                if (value == upperBound - 1)
                    notify();
            }
    }

    public static void main(String[] args)
    {
        Counter c = new Counter(-100, 100);
        Thread p = new Plus(c);
        Thread m = new Moins(c);
        p.start();
        m.start();
    }
}

class Plus extends Thread
{
    private Counter c;

    Plus(Counter c)
    {
        this.c = c;
    }

    public void run()
    {
        while(true)
            try
                {
                    c.increaseCounter();
                }
            catch(InterruptedException e){}
    }
}

class Moins extends Thread
{
    private Counter c;

    Moins(Counter c)
    {
        this.c = c;
    }

    public void run()
    {
        while(true)
            try
                {
                    c.decreaseCounter();
                }
            catch(InterruptedException e){}
    }
}
```

Ce programme affiche aléatoirement les valeurs prises par un compteur incrémenté et décrémenté alternativement par deux threads. Si l'on tente de décrémenter la valeur minimale, le thread de décrémentation s'endort pour laisser la main au thread d'incrémentation. Si le thread d'incrémentation est parti de la valeur minimale, il réveille le thread de décrémentation qui peut reprendre son exécution. Et vice-versa.

Chapitre 2

Exercices

2.1 Le Java procédural

2.1.1 Initiation

Exercice 1 - Compte à rebours

Ecrire un sous-programme prenant en paramètre un nombre n et affichant les valeurs n, $n-1$, ..., 1, 0.

2.1.2 Arithmétique

Exercice 2 - chiffres et nombres

Rapellons que `a % b` est le reste de la division entière de a par b.

1. Ecrire la fonction `static int unites(int n)` retournant le chiffre des unités du nombre n.
2. Ecrire la fonction `static int dizaines(int n)` retournant le chiffre des dizaines du nombre n.
3. Ecrire la fonction `static int extrait(int n, int p)` retourant le p-ème chiffre de représentation décimale de n en partant des unités.
4. Ecrire la fonction `static int nbChiffres(int n)` retournant le nombre de chiffres que comporte la représentation décimale de n.
5. Ecrire la fonction `static int sommeChiffres(int n)` retournant la somme des chiffres de n.

Exercice 3 - Nombres amis

Soient a et b deux entiers strictement positifs. a est un diviseur strict de b si a divise b et $a \neq b$. Par exemple, 3 est un diviseur strict de 6. Mais 6 n'est pas un diviseur strict 6. a et b sont des nombres amis si la somme des diviseurs stricts de a est b et si la somme des diviseurs de b est a. Le plus petit couple de nombres amis connu est 220 et 284.

1. Ecrire une fonction `static int sommeDiviseursStricts(int n)`, elle doit renvoyer la somme des diviseurs stricts de n.
2. Ecrire une fonction `static boolean sontAmis(int a, int b)`, elle doit renvoyer *true* si a et b sont amis, $false$ sinon.

Exercice 4 - Nombres parfaits

Un nombre parfait est un nombre égal à la somme de ses diviseurs stricts. Par exemple, 6 a pour diviseurs stricts 1, 2 et 3, comme $1+2+3=6$, alors 6 est parfait.

1. Est-ce que 18 est parfait ?
2. Est-ce que 28 est parfait ?

3. Que dire d'un nombre ami avec lui-même ?
4. Ecrire la fonction `static boolean estParfait(int n)`, elle doit retourner *true* si n est un nombre parfait, $false$ sinon.

Exercice 5 - Nombres de Kaprekar

Un nombre n est un nombre de Kaprekar en base 10, si la représentation décimale de n^2 peut être séparée en une partie gauche u et une partie droite v tel que $u + v = n$. $45^2 = 2025$, comme $20 + 25 = 45$, 45 est aussi un nombre de Kaprekar. $4879^2 = 23804641$, comme $238 + 04641 = 4879$ (le 0 de 046641 est inutile, je l'ai juste placé pour éviter toute confusion), alors 4879 est encore un nombre de Kaprekar.

1. Est-ce que 9 est un nombre de Kaprekar ?
2. Ecrire la fonction `static int sommeParties(int n, int p)` qui découpe n est deux nombres dont le deuxième comporte p chiffres, et aui retourne leur somme. Par exemple,

$$sommeParties(12540, 2) = 125 + 40 = 165$$

3. Ecrire la fonction `static boolean estKaprekar(int n)`

2.1.3 Pour le sport

Exercice 6 - Conversion chiffres/lettres

Ecrire un sous-programme prenant un nombre en paramètre et l'affichant en toutes lettres. Rappelons que 20 et 100 s'accordent en nombre s'ils ne sont pas suivis d'un autre nombre (ex. : quatre-vingts, quatre-vingt-un). Mille est invariable (ex. : dix-mille) mais pas million (ex. : deux millions) et milliard (ex. : deux milliards). Depuis 1990, tous les mots sont séparés de traits d'union (ex. : quatre-vingt-quatre), sauf autour des mots mille, millions et milliard (ex. : deux mille deux-cent-quarante-quatre, deux millions quatre-cent-mille-deux-cents). (source : `http://www.leconjugueur.com/frlesnombres.php`).

2.2 Objets

2.2.1 Création d'une classe

Exercice 1 - La classe `Rationnel`

Créez une classe `Rationnel` contenant un numérateur et un dénominateur tous deux de type entier. Instanciez deux rationnels a et b et initialisez-les aux valeurs respectives $\frac{1}{2}$ et $\frac{4}{3}$. Affichez ensuite les valeurs de ces champs.

2.2.2 Méthodes

Exercice 2 - Opérations sur les Rationnels

Ajoutez à la classe `Rationnel` les méthodes suivantes :

1. `public String toString()`, retourne une chaîne de caractères contenant une représentation du rationnel courant sous forme de chaîne de caractères.
2. `public Rationnel copy()`, retourne une copie du rationnel courant.
3. `public Rationnel opposite()`, retourne l'opposé du rationnel courant.
4. `public void reduce()`, met le rationnel sous forme de fraction irréductible.
5. `public boolean isPositive()`, retourne `true` si et seulement si le rationnel courant est strictement positif.
6. `public Rationnel add(Rationnel other)`, retourne la somme du rationnel courant et du rationnel `other`.
7. `public void addBis(Rationnel other)`, additionne le rationnel `other` au rationnel courant.
8. `public Rationnel multiply(Rationnel other)`, retourne le produit du rationnel courant avec le rationnel `others`.
9. `public Rationnel divide(Rationnel other)`, retourne le quotient du rationnel courant avec le rationnel `others`.
10. `public int compareTo(Rationnel other)`, retourne 0 si le rationnel courant est égal au rationnel `other`, -1 si le rationnel courant est inférieur à `other`, 1 dans le cas contraire.

2.3 Tableaux

2.3.1 Prise en main

Exercice 1 - Création et affichage

Créer un tableau de 20 entiers, y placer les nombre de 1 à 20, inverser l'ordre des éléments avec une fonction (récursive si vous y arrivez).

2.3.2 Tris

Exercice 2 - Tris

Créer une classe `RatTab` contenant un tableau de rationnels, ajoutez les méthodes suivantes :

1. `public void init(int taille)`, instancie le tableau et y place `taille` valeurs aléatoires `import java.math.Random ;`).
2. `public String toString()`, retourne une chaîne de caractère représentant le tableau.
3. `public void triInsertion()`, tri le tableau par la méthode du tri par insertion.
4. `public void triBulle()`, tri le tableau par la méthode du tri à bulle.
5. `public void triSelection()`, tri le tableau par la méthode du tri par sélection.
6. `public void initTab(RatTab otther)`, place dans le tableau une copie du tableau de `other`.
7. `public RatTab copie()`, retourne une copie de l'instance courante.
8. `public void interclasse(RatTab t1, RatTab t2)`, place dans le tableau l'interclassement des tableaux t1 et t2.

2.4 Encapsulation

2.4.1 Prise en main

Exercice 1 - Rationnels propres

Reprennez la classe `Rationnel` et faites en une classe `RationnelPropre`. Vous encapsulerez le numérateur, le dénominateur, et vous utiliserez un constructeur pour initialiser les champs privés. Vous prendrez ensuite le soin de modifier les corps des méthodes de sorte que la classe compile et fonctionne correctement.

Exercice 2 - Listes chaînées

Complétez le code ci-dessous :

```
public class IntList
{
    /*
      Donnee stockee dans le maillon
     */

    private int data;

    /*********************************/

    /*
      Pointeur vers le maillon suivant
     */

    private IntList next;

    /*********************************/

    /*
      Constructeur initialisant la donnee et
      le pointeur vers l'element suivant.
     */

    IntList(int data, IntList next)
    {
    }

    /*********************************/

    /*
      Constructeur initialisant la donnee et
      mettant le suivant a null.
     */

    IntList(int data)
    {
    }

    /*********************************/

    /*
      Constructeur recopiant tous les maillons de other.
```

```
 */

IntList(IntList other)
{
}

/*********************************/

/*
  Retourne la donnee.
 */

public int getData()
{
    return 0;
}
/*********************************/

/*
  Modifie la donnee
 */

public void setData(int data)
{
}

/*********************************/

/*
  Retourne le maillon suivant.
 */

public IntList getNext()
{
    return null;
}

/*********************************/

/*
  Modifie le maillon suivant
 */

public void setNext(IntList next)
{
}

/*********************************/

/*
  Retourne une reprÃ©sentation sous forme de
  chaine de la liste.
 */
```

```
    public String toString()
    {
        return null;
    }

    /*********************************/

    /*
      Teste le fonctionnement de la liste.
     */

    public static void main(String[] args)
    {
        IntList l = new IntList(20);
        int i = 19;
        while (i >= 0)
            l = new IntList(i--, l);
        System.out.println(l);

    }

}
```

2.4.2 Implémentation d'une Pile

Exercice 3 - Avec un tableau

Complétez le code ci-dessous :

```
public class Pile
{
    /*
        Tableau contenant les elements de la pile.
    */

    private int[] tab;

    /*****************************************/

    /*
        Taille de la pile
    */

    private final int taille;

    /*****************************************/

    /*
        Indice du premier element non occupe dans
        le tableau.
    */

    private int firstFree;

    /*****************************************/
```

```
/*
    Constructeur
*/

Pile(int taille)
{
    this.taille = 0;
}

/*****************************************/

/*
    Constructeur de copie
*/

Pile(Pile other)
{
    this(other.taille);
}

/*****************************************/

/*
    Retourne vrai si et seulement
    si la pile est vide
*/

public boolean estVide()
{
    return true;
}

/*****************************************/

/*
    Retourne vrai si et seulement si la pile est
    pleine.
*/

public boolean estPleine()
{
    return true;
}

/*****************************************/

/*
    Retourne l'element se trouvant au sommet de
    la pile, -1 si la pile est vide.
*/

public int sommet()
{
```

```
        return 0;
    }

    /*****************************************/

    /*
        Supprime l'element se trouvant au sommet
        de la pile, ne fait rien si la pile est
        vide.
    */

    public void depile()
    {
    }

    /*****************************************/

    /*
        Ajoute data en haut de la pile, ne fait rien
        si la pile est pleine.
    */

    public void empile(int data)
    {
    }

    /*****************************************/

    /*
        Retourne une representation de la pile
        au format chaine de caracteres.
    */

    public String toString()
    {
        return null;
    }

    /*****************************************/

    /*
        Teste le fonctionnement de la pile.
    */

    public static void main(String[] args)
    {
        Pile p = new Pile (30);
        int i = 0;
        while(!p.estPleine())
            p.empile(i++);
        System.out.println(p);
        while(!p.estVide())
            {
                System.out.println(p.sommet());
```

```
                p.depile();
            }
        }
}
```

Exercice 4 - Avec une liste chaînée

Re-téléchargez le fichier `Pile.java`, et sans modifier les méthodes publiques, implémentez la pile en utilisant des listes chaînées (`IntList`).

2.5 Héritage

2.5.1 Héritage

Exercice 1 - Point

Définir une classe `Point` permettant de représenter un point dans R^2. Vous utiliserez proprement les concepts d'encapsulation.

Exercice 2 - Cercle

Définir une classe `Cercle` héritant de de la classe précédente. Vous prendrez soin de ne pas recoder des choses déjà codées...

Exercice 3 - Pile

Adaptez la pile pour qu'elle puisse contenir des objets de n'importe quel type.

Exercice 4 - Redéfinition de méthode

Reprennez les classes `Point` et `Cercle`, codez une méthode `toString()` pour les deux classes (mère et fille). Vous prendrez soin d'éviter les redondances dans votre code.

2.5.2 Interfaces

Exercice 5 - Animaux

Téléchargez l'archive `animaux.tgz` (depuis le site seulement), et complétez le code source, sachant qu'un chien dit "Ouaf !", un chat "Miaou !" et une vache "Meuh !".

Exercice 6 - Tab

Reprennez la classe RatTab et adaptez-là de sorte qu'on puisse y placer tout objet d'un type implémentant l'interface prédéfinie `Comparable`.

2.5.3 Classes abstraites

Exercice 7 - Animaux

Reprennez le tp sur les animaux, et faites remonter les méthodes `setNom` et `getNom` dans la classe mère.

Exercice 8 - Arbres syntaxiques

Téléchargez l'archive `arbresSyntaxiques.tgz` (depuis le site seulement), complétez le code source.

2.6 Exceptions

Exercice 1 - l'exception inutile

Créer une classe `ClasseInutile` contenant une seule méthode `nePasInvoquer()` contenant une seule instruction, qui lève l'exception `ExceptionInutile`. Lorsqu'elle est levée, l'`ExceptionInutile` affiche "Je vous avais dit de ne pas invoquer cette fonction !". Vous testerez la méthode `nePasInvoquer()` dans le `main` de `ClasseInutile`.

Exercice 2 - Pile

Reprennez le code de la pile, modifiez-le de sorte que si l'on tente de dépiler une pile vide, une exception soit levée.

2.7 Interfaces graphiques

2.7.1 Prise en main

Exercice 1 - Ecouteur d'événement

Créer une fenêtre "Gestionnaire de disque dur". Ajouter un bouton ayant pour intitulé "Formater le disque dur" et affichant dans la console le message "formatage en cours". Vous utiliserez les trois méthodes suivantes :

1. La même que dans le premier exemple du cours : impléméntation de l'écouteur d'événement par la `JFrame`.
2. Avec une classe anonyme.
3. Avec une classe non anonyme.

Exercice 2 - Gestionnaires de mises en forme

Modifier le programme de l'exercice précédent en utilisant un `GridLayout`. Vous afficherez le message "formatage en cours" non pas sur la console mais dans une zone de texte.

2.7.2 Maintenant débrouillez-vous

Exercice 3 - Convertisseur

Coder un convertisseur euro/dollar.

Exercice 4 - Calculatrice

Coder une calculatrice.

2.8 Collections

2.8.1 Packages

Exercice 1 - Pile

Récupérer le code source de pile, implémentée avec des listes chaînées, pour créer un package portant votre prénom. Vous vérifierez votre installation en utilisant la classe pile en dehors du package. Vous déterminerez intelligemment la visibilité des divers objets.

2.8.2 Types paramétrées

Exercice 2 - Héritage

Vérifiez expérimentalement si `ExempleClasseParametree<String>` hérite de `ExempleClasseParametree<Object>`. Trouvez une explication...

Exercice 3 - Types paramétrés et interfaces

Créez une classe `DeuxObjets` contenant deux objets de même type `<T>` et deux méthodes `getPetit` et `getGrand` retournant respectivement le plus petit des deux objets et le plus grand.

Exercice 4 - Pile

Reprennez le package sur la pile et adaptez la pour que la pile, implémentée avec des listes chaînées puisse fonctionner avec des objets de n'importe quel type.

2.8.3 Collections

Exercice 5 - Parcours

Créez une `ArrayList<T>`, placez-y des valeurs quelconques, et parcourez-là en affichant ses valeurs.

Exercice 6 - `Iterable<T>`

Modifiez la Pile pour qu'elle implémente `Iterable<T>`. Adaptez la fonction `toString()` en conséquence.

Exercice 7 - Parcours

Créez un `TreeSet<T>`, placez-y des `DeuxObjets<Integer>`, et parcourez-le en affichant ses valeurs.

2.9 Threads

2.9.1 Prise en main

Exercice 1 - La méthode `start()`

Remplacez les appels à `start()` par `run()`, que remarquez-vous ?

Exercice 2 - `Runnable`

Reprennez le code de `BinaireAleatoire` et adaptez-le pour utiliser l'interface `Runnable`.

2.9.2 Synchronisation

Exercice 3 - Méthode synchronisée

Créez un compteur commun aux deux threads `un` et `zero`, vous y placerez une fonction `synchronized boolean stepCounter()` qui retournera vrai tant que le compteur n'aura pas atteint sa limite, et qui retournera tout le temps faux ensuite. Le but étant d'afficher 30000 chiffres, peu importe que ce soit des 1 ou des 0.

Exercice 4 - Instructions synchronisées

Reprennez l'exercice précédent en synchronisant les appels à `stepCounter()` dans la méthode `run()`.

2.9.3 Débrouillez-vous

Exercice 5 - Producteur consommateur

Implémentez une file permettant des accès synchronisés. Créez une classe écrivant des valeurs aléatoires dans la file et une classe lisant ces valeurs et les affichant au fur et à mesure. Lancez plusiseurs instances de la classe de lecture et plusieurs instances de la classe d'écriture sur la même file.

Exercice 6 - Diner des philosophes

Implémentez le problème du diner des philosophes. Faites en sorte qu'il n'y ait ni famine ni interblocage.

Chapitre 3

Corrigés

3.1 Java Procédural

3.1.1 Initiation

```
class CountDown
{
    static void countDown(int n)
    {
        while(n >= 0)
            {
                System.out.println(n);
                n--;
            }
    }

    public static void main(String[] args)
    {
        countDown(20);
    }
}
```

3.1.2 Arithmétique

```
class Arithmetique
{
    static int puissance(int b, int n)
    {
        int res = 1;
        for (int i = 1 ; i <= n ; i++)
            res *= b;
        return res;
    }

    static int unites(int n)
    {
        return n%10;
    }

    static int dizaines(int n)
    {
        return (n%100)/10;
    }

    static int extrait(int n, int p)
    {
        return (n % puissance(10, p))/puissance(10, p-1);
    }

    public static int nbChiffres(int n)
    {
        int res = 0;
        while(n != 0)
            {
                res ++;
                n/=10;
            }
```

```
        return res;
    }

    static int sommeChiffres(int n)
    {
        int nbC = nbChiffres(n);
        int somme = 0;
        for(int i = 1 ; i <= nbC ; i++)
            somme += extrait(n, i);
        return somme;
    }

    static boolean divise(int a, int b)
    {
        return b%a == 0;
    }

    static int sommeDiviseursStricts(int n)
    {
        int somme = 0;
        for(int i = 1 ; i < n ; i++)
            if(divise(i, n))
                somme += i;
        return somme;
    }

    static boolean sontAmis(int a, int b)
    {
        return sommeDiviseursStricts(a) == sommeDiviseursStricts(b);
    }

    static boolean estParfait(int n)
    {
        return n == sommeDiviseursStricts(n);
    }

    static int sommeParties(int n, int p)
    {
        return n%puissance(10, p) + n/puissance(10, p);
    }

    static boolean estKaprekar(int n)
    {
        int carre = n*n;
        int nbc = nbChiffres(n*n);
        for(int i = 1; i < nbc ; i++)
            if (sommeParties(carre, i) == n)
                return true;
        return false;
    }

    public static void main(String[] args)
    {
        System.out.println(estKaprekar(3456789));
        System.out.println(estKaprekar(45));
        System.out.println(estKaprekar(4879));
    }
}
```

3.1.3 Pour le sport

```
class StringOfInt
{

    /*
      Affiche le nombre passe en parametre s'il est compris entre 1 et 19.
     */

    static void afficheUnites(int n)
    {
        switch(n)
            {
            case 1 : System.out.print("un"); break;
            case 2 : System.out.print("deux"); break;
            case 3 : System.out.print("trois"); break;
            case 4 : System.out.print("quatre"); break;
            case 5 : System.out.print("cinq"); break;
            case 6 : System.out.print("six"); break;
            case 7 : System.out.print("sept"); break;
            case 8 : System.out.print("huit"); break;
            case 9 : System.out.print("neuf"); break;
```

```
        case 10 : System.out.print("dix"); break;
        case 11 : System.out.print("onze"); break;
        case 12 : System.out.print("douze"); break;
        case 13 : System.out.print("treize"); break;
        case 14 : System.out.print("quatorze"); break;
        case 15 : System.out.print("quinze"); break;
        case 16 : System.out.print("seize"); break;
        case 17 : System.out.print("dix-sept"); break;
        case 18 : System.out.print("dix-huit"); break;
        case 19 : System.out.print("dix-neuf"); break;
        default : break;
        }
}

/*
  Affiche les dizaines de 10*n si n est compris entre 2 et 9.
 */

static void afficheDizaines(int n)
{
    switch(n)
        {
        case 2 : System.out.print("vingt"); break;
        case 3 : System.out.print("trente"); break;
        case 4 : System.out.print("quarante"); break;
        case 5 : System.out.print("cinquante"); break;
        case 6 :
        case 7 : System.out.print("soixante"); break;
        case 8 :
        case 9 : System.out.print("quatre-vingt"); break;
        default : break;
        }
}

/*
  Retourne b^n
 */

static int puissance(int b, int n)
{
    int res = 1;
    for(int i = 1 ; i <= n ; i++)
        res *= b;
    return res;
}

/*
  Extrait les chiffres de n par tranche. On indices ces chiffres a partir de 1
  en partant de la droite. debut est l'indice du debut de la tranche et fin
  l'indice de la fin. Par exemple extraitTranche(987654321, 5, 2) = 5432.
 */

static int extraitTranche(int n, int debut, int fin)
{
    return (n%puissance(10, debut))/puissance(10, fin - 1);
}

/*
  Affiche en toutes lettres le nombre n, n doit etre compris entre
  0 et 999.
 */

static void afficheTroisChiffres(int n)
{
    int unites = extraitTranche(n, 1, 1);
    int dizaines = extraitTranche(n, 2, 2);
    int dizainesUnites = extraitTranche(n, 2, 1);
    int centaines = extraitTranche(n, 3, 3);
    if (centaines >= 2)
        {
            afficheUnites(centaines);
            System.out.print("-");
        }
    if (centaines != 0)
        System.out.print("cent");
    if (dizainesUnites == 0 && centaines != 1 && centaines != 0)
        System.out.print("s");
    if (dizainesUnites != 0 && centaines != 0)
        System.out.print("-");
    afficheDizaines(dizaines);
    if (unites == 0 && dizaines == 8)
```

```
            System.out.print("s");
        if (dizaines != 0 && dizaines != 1 &&
            (unites != 0 || dizaines == 7 || dizaines == 9))
            System.out.print("-");
        if (unites == 1 && dizaines >= 2 && dizaines <= 6)
            System.out.print("et-");
        if(dizaines == 1 || dizaines == 7 || dizaines == 9)
            afficheUnites((dizainesUnites - 10 * (dizaines - 1)));
        else
            afficheUnites(unites);
    }

    /*
      Affiche le nombre n suivi de la chaine unite, accorde
      est vrai si l'unite n'est pas invariable.
     */
    static void afficheNombreEtUnite(int n, String unite, boolean accorde)
    {
        if (accorde || n != 1)
            {
                afficheTroisChiffres(n);
                if (n != 0)
                    System.out.print(" ");
            }
        System.out.print(unite);
        if (accorde && n > 1)
            System.out.print("s");
    }

    static void afficheNombre(int n)
    {
        int billions = extraitTranche(n, 15, 13);
        int milliards = extraitTranche(n, 12, 10);
        int millions = extraitTranche(n, 9, 7);
        int milliers = extraitTranche(n, 6, 4);
        int unites = extraitTranche(n, 3, 1);
        if (billions != 0)
            {
                afficheNombreEtUnite(billions, "billion", true);
                System.out.print(" ");
            }
        if (milliards != 0)
            {
                afficheNombreEtUnite(milliards, "milliard", true);
                System.out.print(" ");
            }
        if (millions != 0)
            {
                afficheNombreEtUnite(millions, "million", true);
                System.out.print(" ");
            }
        if (milliers != 0)
            {
                afficheNombreEtUnite(milliers, "mille", false);
                System.out.print(" ");
            }
        if (unites != 0)
            {
                afficheNombreEtUnite(unites, "", false);
            }
    }

    public static void main(String[] args)
    {
        for(int i = 1 ; i <= 30 ; i++)
            {
                System.out.print(puissance(3, i) + " = ");
                afficheNombre(puissance(3, i));
                System.out.println("");
            }
    }
}
```

3.2 Objets

3.2.1 Création d'une classe

La classe `Rationnel`

```
public class Rationnel
{
    int num, den;

    public static void main(String[] args)
    {
        Rationnel a, b;
        a = new Rationnel();
        b = new Rationnel();
        a.num = 1;
        a.den = 2;
        b.num = 4;
        b.den = 3;
        System.out.println("a = " + a.num + "/" + a.den);
        System.out.println("b = " + b.num + "/" + b.den);
    }
}
```

3.2.2 Méthodes

Opérations sur les Rationnels

```
public class Rationnel
{
    int num, den;

    /*-------------------------------------------------------*/

    public String toString()
    {
        return num + "/" + den;
    }

    /*-------------------------------------------------------*/

    public Rationnel copy()
    {
        Rationnel r = new Rationnel();
        r.num = num;
        r.den = den;
        return r;
    }

    /*-------------------------------------------------------*/

    public Rationnel opposite()
    {
        Rationnel r = copy();
        r.num = -r.num;
        return r;
    }

    /*-------------------------------------------------------*/

    private int pgcd(int a, int b)
    {
        if (b == 0)
            return a;
        return pgcd(b, a%b);
    }

    public void reduce()
    {
        int p = pgcd(num, den);
        num/=p;
        den*=p;
    }

    /*-------------------------------------------------------*/

    public boolean isPositive()
    {
        return num > 0 && den > 0 || num < 0 && den < 0;
    }

    /*-------------------------------------------------------*/

    public Rationnel add(Rationnel other)
    {
        Rationnel res = new Rationnel();
```

```
        res.num = num*other.den + den*other.num;
        res.den = den * other.den ;
        other.reduce();
        return res;
    }

    /*-------------------------------------------------------*/

    public void addBis(Rationnel other)
    {
        num = num*other.den + den*other.num;
        den = den * other.den ;
        reduce();
    }

    /*-------------------------------------------------------*/

    public Rationnel multiply(Rationnel other)
    {
        Rationnel res = new Rationnel();
        res.num = num*other.num;
        res.den = den * other.den ;
        other.reduce();
        return res;
    }

    /*-------------------------------------------------------*/

    public Rationnel divide(Rationnel other)
    {
        Rationnel res = new Rationnel();
        res.num = num*other.den;
        res.den = den * other.num ;
        other.reduce();
        return res;
    }

    /*-------------------------------------------------------*/

    public int compareTo(Rationnel other)
    {
        Rationnel sub = add(other.opposite());
        if (sub.isPositive())
            return 1;
        if (sub.opposite().isPositive())
            return -1;
        return 0;
    }

    /*-------------------------------------------------------*/

    public static void main(String[] args)
    {
        Rationnel a, b;
        a = new Rationnel();
        b = new Rationnel();
        a.num = 1;
        a.den = 2;
        b.num = 4;
        b.den = 3;
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("compareTo(" + a + ", " +
                           b + ") = " + a.compareTo(b));
        System.out.println(a.copy());
        System.out.println(a.opposite());
        System.out.println(a.add(b));
        System.out.println(a.multiply(b));
        System.out.println(a.divide(b));

    }
}
```

3.3 Tableaux

3.3.1 Prise en main

Création et affichage

```
public class PriseEnMainTableaux
{

    /*-------------------------------------------------------*/

    public static void affiche(int[] t)
    {
        for (int i = 0 ; i < t.length ; i++)
            {
                System.out.print(t[i] + " ");
            }
        System.out.println();
    }

    /*-------------------------------------------------------*/

    public static void swap(int[] t, int i, int j)
    {
        int temp = t[i];
        t[i] = t[j];
        t[j] = temp;
    }

    /*-------------------------------------------------------*/

    public static void reverse(int[] t)
    {
        int i = 0, j = t.length - 1;
        while(i < j)
            swap(t, i++, j--);
    }

    /*-------------------------------------------------------*/

    public static void reverseRec(int[] t, int i, int j)
    {
        if (i < j)
            {
                swap(t, i, j);
                reverseRec(t, i + 1, j - 1);
            }
    }

    /*-------------------------------------------------------*/

    public static void reverseBis(int[] t)
    {
        reverseRec(t, 0, t.length - 1);
    }

    /*-------------------------------------------------------*/

    public static void main(String[] args)
    {
        int[] t = new int[20];
        for (int i = 0 ; i < 20 ; i++)
            t[i] = i;
        affiche(t);
        reverse(t);
        affiche(t);
        reverseBis(t);
        affiche(t);
    }
}
```

3.3.2 Tris

Tris

```
import java.util.Random;
import java.lang.Math;

public class RatTab
{
    Rationnel[] t;

    /*-------------------------------------------------*/

    public void init(int taille)
    {
```

```
        t = new Rationnel[taille];
        Random r = new Random();
        for (int i = 0 ; i < taille ; i++)
            {
                t[i] = new Rationnel();
                t[i].num = r.nextInt()%10;
                t[i].den = Math.abs(r.nextInt()%10)+1;
            }
    }

    /*-------------------------------------------------*/

    public String toString()
    {
        String res = "[";
        if (t.length >= 1)
            res += t[0];
        for (int i = 1 ; i < t.length ; i++)
            res += ", " + t[i];
        res += "]";
        return res;
    }

    /*-------------------------------------------------*/

    public void echange(int i, int j)
    {
        Rationnel temp = t[i];
        t[i] = t[j];
        t[j] = temp;
    }

    /*-------------------------------------------------*/

    public void triSelection()
    {
        for (int i = 0 ; i < t.length - 1 ; i++)
            {
                int indiceMin = i;
                for (int j = i + 1 ; j < t.length; j++)
                    if (t[indiceMin].compareTo(t[j]) > 1)
                        indiceMin = j;
                echange(i, indiceMin);
            }
    }

    /*-------------------------------------------------*/

    public boolean ordonne(int i, int j)
    {
        if ((i - j) * (t[i].compareTo(t[j])) < 0)
            {
                echange(i, j);
                return true;
            }
        return false;
    }

    /*-------------------------------------------------*/

    public void triInsertion()
    {
        for (int i = 1 ; i < t.length ; i++)
            {
                int j = i;
                while(j > 0 && ordonne(j-1, j))
                    j--;
            }
    }

    /*-------------------------------------------------*/

    public void triBulle()
    {
        for (int i = t.length - 1 ; i > 0 ; i--)
            for (int j = 0 ; j < i ; j++)
                ordonne(j, j+1);
    }

    /*-------------------------------------------------*/
```

```
    public void initTab(RatTab other)
    {
        t = new Rationnel[other.t.length];
        for (int i = 0 ; i < t.length ; i++)
            t[i] = other.t[i];
    }

    /*-------------------------------------------------*/

    public RatTab copie()
    {
        RatTab other = new RatTab();
        other.initTab(this);
        return other;
    }

    /*-------------------------------------------------*/

    public void interclasse(RatTab t1, RatTab t2)
    {
        t = new Rationnel[t1.t.length + t2.t.length];
        int i1 = 0, i2 = 0, i = 0;
        while(i < t.length)
            {
                if (i1 == t1.t.length)
                        t[i++] = t2.t[i2++].copy();
                else if (i2 == t2.t.length)
                        t[i++] = t1.t[i1++].copy();
                else if (t1.t[i1].compareTo(t2.t[i2])<0)
                        t[i++] = t1.t[i1++].copy();
                else
                        t[i++] = t2.t[i2++].copy();
            }
    }

    /*-------------------------------------------------*/

    public static void main(String[] args)
    {
        RatTab t = new RatTab();
        t.init(10);
        System.out.println(t);
        t.triInsertion();
        System.out.println(t);
        t.triSelection();
        System.out.println(t);
        t.triBulle();
        System.out.println(t);
        RatTab tBis = new RatTab();
        tBis.initTab(t);
        System.out.println(tBis);
        tBis = t.copie();
        System.out.println(tBis);
        tBis.init(10);
        System.out.println(tBis);
        RatTab tTer = new RatTab();
        tBis.triInsertion();
        System.out.println(tBis);
        tTer.interclasse(t, tBis);
        System.out.println(tTer);
        tTer.triInsertion();
        System.out.println(tTer);
    }
}
```

3.4 Encapsulation

3.4.1 Prise en main

Rationnels propres

```
public class RationnelPropre
{
    private int num, den;

    /*-------------------------------------------------*/

    public RationnelPropre(int num, int den)
    {
```

```
    this.num = num;
    this.den = den;
}

/*-------------------------------------------------------*/

public int getNum()
{
    return num;
}

/*-------------------------------------------------------*/

public int getDen()
{
    return den;
}

/*-------------------------------------------------------*/

public void setNum(int num)
{
    this.num = num;
}

/*-------------------------------------------------------*/

public void setDen(int den)
{
    if (den != 0)
        this.den = den;
    else
        System.out.println("Division by Zero !!!");
}

/*-------------------------------------------------------*/

public String toString()
{
    return num + "/" + den;
}

/*-------------------------------------------------------*/

public RationnelPropre copy()
{
    RationnelPropre r = new RationnelPropre(num, den);
    return r;
}

/*-------------------------------------------------------*/

public RationnelPropre opposite()
{
    RationnelPropre r = copy();
    r.setNum(-r.getNum());
    return r;
}

/*-------------------------------------------------------*/

private int pgcd(int a, int b)
{
    if (b == 0)
        return a;
    return pgcd(b, a%b);
}

public void reduce()
{
    int p = pgcd(num, den);
    num/=p;
    den*=p;
}

/*-------------------------------------------------------*/

public boolean isPositive()
{
    return num > 0 && den > 0 || num < 0 && den < 0;
}
```

```
    /*-------------------------------------------------*/

    public RationnelPropre add(RationnelPropre other)
    {
        RationnelPropre res =
            new RationnelPropre(
                                num*other.getDen() + den*other.getNum(),
                                den * other.getDen());
        other.reduce();
        return res;
    }

    /*-------------------------------------------------*/

    public void addBis(RationnelPropre other)
    {
        num = num*other.getDen() + den*other.getNum();
        den = den * other.getDen() ;
        reduce();
    }

    /*-------------------------------------------------*/

    public RationnelPropre multiply(RationnelPropre other)
    {
        RationnelPropre res = new RationnelPropre(num*other.getNum(),
                                                  den * other.getDen());
        other.reduce();
        return res;
    }

    /*-------------------------------------------------*/

    public RationnelPropre divide(RationnelPropre other)
    {
        RationnelPropre res = new RationnelPropre(num*other.getDen(),
                                                  den * other.getNum());
        other.reduce();
        return res;
    }

    /*-------------------------------------------------*/

    public int compareTo(RationnelPropre other)
    {
        RationnelPropre sub = add(other.opposite());
        if (sub.isPositive())
            return 1;
        if (sub.opposite().isPositive())
            return -1;
        return 0;
    }

    /*-------------------------------------------------*/

    public static void main(String[] args)
    {
        RationnelPropre a, b;
        a = new RationnelPropre(1, 2);
        b = new RationnelPropre(3, 4);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("compareTo(" + a + ", " +
                           b + ") = " + a.compareTo(b));
        System.out.println(a.copy());
        System.out.println(a.opposite());
        System.out.println(a.add(b));
        System.out.println(a.multiply(b));
        System.out.println(a.divide(b));

    }
}
```

Listes chaînées

```
public class IntListCorrige
{
    /*
      Donnee stockee dans le maillon
     */
```

```
private int data;

/*********************************/

/*
  Pointeur vers le maillon suivant
 */

private IntListCorrige next;

/*********************************/

/*
  Constructeur initialisant la donnee et
  le pointeur vers l'element suivant.
 */

IntListCorrige(int data, IntListCorrige next)
{
    this.data = data;
    this.next = next;
}

/*********************************/

/*
  Constructeur initialisant la donnee et
  mettant le suivant a null.
 */

IntListCorrige(int data)
{
    this(data, null);
}

/*********************************/

/*
  Constructeur initialisant la donnee et
  mettant le suivant a null.
 */

IntListCorrige(IntListCorrige other)
{
    this(other.getData(), new IntListCorrige(other.getNext()));
}

/*********************************/

/*
  Retourne la donnee.
 */

public int getData()
{
    return data;
}
/*********************************/

/*
  Modifie la donnee
 */

public void setData(int data)
{
     this.data = data;
}
/*********************************/

/*
  Retourne le maillon suivant.
 */

public IntListCorrige getNext()
{
    return next;
}

/*********************************/
```

```
    /*
      Modifie le maillon suivant
     */

    public void setNext(IntListCorrige next)
    {
        this.next = next;
    }

    /*******************************/

    /*
      Retourne une reprÃ©sentation sous forme de
      chaine de la liste.
     */

    public String toString()
    {
        String res = "" + data;
        if (next != null)
            res += " -> " + next.toString();
        return res;
    }

    /*******************************/

    /*
      Teste le fonctionnement de la liste.
     */

    public static void main(String[] args)
    {
        IntListCorrige l = new IntListCorrige(20);
        int i = 19;
        while (i >= 0)
            l = new IntListCorrige(i--, l);
        System.out.println(l);
    }

}
```

3.4.2 Implémentation d'une pile

Avec un tableau

```
public class PileCorrige
{
    /*
        Tableau contenant les elements de la pile.
    */

    private int[] tab;

    /*****************************************/

    /*
        Taille de la pile
    */

    private final int taille;

    /*****************************************/

    /*
        Indice du premier element non occupe dans
        le tableau.
    */

    private int firstFree;

     /*****************************************/

    /*
        Constructeur
    */

    PileCorrige(int taille)
    {
        this.taille = taille;
        tab = new int[taille];
```

```
    firstFree = 0;
}

 /*****************************************/

/*
    Constructeur de copie.
*/

PileCorrige(PileCorrige other)
{
    this(other.taille);
    firstFree = other.firstFree;
    for(int i = 0 ; i < firstFree ; i++)
        tab[i] = other.tab[i];
}

/*****************************************/

/*
    Retourne vrai si et seulement
    si la pile est vide
*/

public boolean estVide()
{
    return firstFree == 0;
}

/*****************************************/

/*
    Retourne vrai si et seulement si la pile est
    pleine.
*/

public boolean estPleine()
{
    return firstFree == taille;
}

/*****************************************/

/*
    Retourne l'element se trouvant au sommet de
    la pile, -1 si la pile est vide.
*/

public int sommet()
{
    if (estVide())
        return -1;
    return tab[firstFree - 1];
}

/*****************************************/

/*
    Supprime l'element se trouvant au sommet
    de la pile, ne fait rien si la pile est
    vide.
*/

public void depile()
{
    if (!estVide())
        firstFree--;
}

/*****************************************/

/*
    Ajoute data en haut de la pile, ne fait rien
    si la pile est pleine.
*/

public void empile(int data)
{
    if (!estPleine())
        tab[firstFree++] = data;
}
```

```
    /*****************************************/

    /*
        Retourne une representation de la pile
        au format chaine de caracteres.
    */

    public String toString()
    {
        String res = "[";
        for (int i = 0 ; i < firstFree ; i++)
            res += " " + tab[i];
        return res + " ]";
    }

    /*****************************************/

    /*
        Teste le fonctionnement de la pile.
    */

    public static void main(String[] args)
    {
        PileCorrige p = new PileCorrige (30);
        int i = 0;
        while(!p.estPleine())
            p.empile(i++);
        System.out.println(p);
        while(!p.estVide())
            {
                System.out.println(p.sommet());
                p.depile();
            }
    }
}
```

Avec une liste chaînée

```
public class PileIntList
{
    /*
        Liste contenant les elements de la pile.
    */

    private IntListCorrige l;

    /*****************************************/

    /*
        Taille de la pile
    */

    private final int taille;

    /*****************************************/

    /*
        Nombre d'elements dans la liste
    */

    private int nbItems;

    /*****************************************/

    /*
        Constructeur
    */

    PileIntList(int taille)
    {
        l = null;
        this.taille = taille;
        nbItems = 0;
    }

    /*****************************************/

    /*
        Constructeur de copie.
    */
```

```
PileIntList(PileIntList other)
{
    taille = other.taille;
    l = new IntListCorrige(other.l);
    nbItems = other.nbItems;
}

/*****************************************/

/*
    Retourne vrai si et seulement
    si la pile est vide
*/

public boolean estVide()
{
    return nbItems == 0;
}

/*****************************************/

/*
    Retourne vrai si et seulement si la pile est
    pleine.
*/

public boolean estPleine()
{
    return nbItems == taille;
}

/*****************************************/

/*
    Retourne l'element se trouvant au sommet de
    la pile, -1 si la pile est vide.
*/

public int sommet()
{
    if (!estVide())
        return l.getData();
    return -1;
}

/*****************************************/

/*
    Supprime l'element se trouvant au sommet
    de la pile, ne fait rien si la pile est
    vide.
*/

public void depile()
{
    if (!estVide())
        {
            l = l.getNext();
            nbItems--;
        }
}

/*****************************************/

/*
    Ajoute data en haut de la pile, ne fait rien
    si la pile est pleine.
*/

public void empile(int data)
{
    l = new IntListCorrige(data, l);
    nbItems ++;
}

/*****************************************/

/*
    Retourne une representation de la pile
    au format chaine de caracteres.
```

```
    */
    public String toString()
    {
        return l.toString();
    }

    /*****************************************/

    /*
        Teste le fonctionnement de la pile.
    */
    public static void main(String[] args)
    {
        PileIntList p = new PileIntList (30);
        int i = 0;
        while(!p.estPleine())
            p.empile(i++);
        System.out.println(p);
        while(!p.estVide())
            {
                System.out.println(p.sommet());
                p.depile();
            }
    }
}
```

3.5 Héritage

3.5.1 Héritage

Point

```
public class PointCorrige
{
    private double abs, ord;

    PointCorrige(double abs, double ord)
    {
        this.abs = abs;
        this.ord = ord;
    }

    PointCorrige()
    {
        this(0, 0);
    }

    public double getAbs()
    {
        return abs;
    }

    public double getOrd()
    {
        return ord;
    }

    public void setAbs(double abs)
    {
        this.abs = abs;
    }

    public void setOrd(double ord)
    {
        this.ord = ord;
    }

    public String toString()
    {
        return "(" + abs + ", " + ord + ")";
    }
}
```

Cercle

```
public class CercleCorrige extends PointCorrige
```

```
{
    private double radius;

    CercleCorrige(double abs, double ord, double radius)
    {
        super(abs, ord);
        this.radius = radius;
    }

    CercleCorrige()
    {
        this(0, 0, 0);
    }

    public double getRadius()
    {
        return radius;
    }

    public void setRadius(double abs)
    {
        this.radius = radius;
    }
    public String toString()
    {
        return "(" + super.toString()+  ", " + radius + ")";
    }
}
```

Liste d'objets

```
public class ObjectList
{
    /*
      Donnee stockee dans le maillon
     */

    private Object data;

    /*********************************/

    /*
      Pointeur vers le maillon suivant
     */

    private ObjectList next;

    /*********************************/

    /*
      Constructeur initialisant la donnee et
      le pointeur vers l'element suivant.
     */

    ObjectList(Object data, ObjectList next)
    {
        this.data = data;
        this.next = next;
    }

    /*********************************/

    /*
      Constructeur initialisant la donnee et
      mettant le suivant a null.
     */

    ObjectList(Object data)
    {
        this(data, null);
    }

    /*********************************/

    /*
      Retourne la donnee.
     */

    public Object getData()
    {
        return data;
```

```
    }
    /*********************************/

    /*
      Modifie la donnee
     */
    public void setData(Object data)
    {
        this.data = data;
    }
    /*********************************/

    /*
      Retourne le maillon suivant.
     */
    public ObjectList getNext()
    {
        return next;
    }

    /*********************************/

    /*
      Modifie le maillon suivant
     */
    public void setNext(ObjectList next)
    {
        this.next = next;
    }

    /*********************************/

    /*
      Retourne une reprÃ©sentation sous forme de
      chaine de la liste.
     */
    public String toString()
    {
        String res = "" + data;
        if (next != null)
            res += " -> " + next.toString();
        return res;
    }

    /*********************************/

    /*
      Teste le fonctionnement de la liste.
     */
    public static void main(String[] args)
    {
        ObjectList l = new ObjectList(20);
        int i = 3;
        while (i >= 0)
            l = new ObjectList(new Integer(i), l);
        System.out.println(l);
    }

}
```

Pile d'objets

```
public class ObjectStack
{
    /*
        Liste contenant les elements de la pile.
    */
    private ObjectList l;

    /*****************************************/

    /*
        Taille de la pile
    */
```

```
private final int taille;

/*****************************************/

/*
    Nombre d'elements dans la liste
*/

private int nbItems;

/*****************************************/

/*
    Constructeur
*/

ObjectStack(int taille)
{
    l = null;
    this.taille = taille;
    nbItems = 0;
}

/*****************************************/

/*
    Retourne vrai si et seulement
    si la pile est vide
*/

public boolean estVide()
{
    return nbItems == 0;
}

/*****************************************/

/*
    Retourne vrai si et seulement si la pile est
    pleine.
*/

public boolean estPleine()
{
    return nbItems == taille;
}

/*****************************************/

/*
    Retourne l'element se trouvant au sommet de
    la pile, -1 si la pile est vide.
*/

public Object sommet()
{
    if (!estVide())
        return l.getData();
    return -1;
}

/*****************************************/

/*
    Supprime l'element se trouvant au sommet
    de la pile, ne fait rien si la pile est
    vide.
*/

public void depile()
{
    if (!estVide())
        {
            l = l.getNext();
            nbItems--;
        }
}

/*****************************************/

/*
```

```
        Ajoute data en haut de la pile, ne fait rien
        si la pile est pleine.
    */
    public void empile(int data)
    {
        l = new ObjectList(data, l);
        nbItems ++;
    }

    /*****************************************/

    /*
        Retourne une representation de la pile
        au format chaine de caracteres.
    */
    public String toString()
    {
        return l.toString();
    }

    /*****************************************/

    /*
        Teste le fonctionnement de la pile.
    */
    public static void main(String[] args)
    {
        ObjectStack p = new ObjectStack (30);
        int i = 0;
        while(!p.estPleine())
            p.empile(new Integer(i++));
        System.out.println(p);
        while(!p.estVide())
            {
                System.out.println(p.sommet());
                p.depile();
            }
    }
}
```

3.5.2 Interfaces

Animaux

```
public interface Animal
{
    public void setNom(String nom);
    public String getNom();
    public String parle();
}

public class Chien implements Animal
{
    private String nom;

    public Chien(String nom)
    {
        this.nom = nom;
    }

    public void setNom(String nom)
    {
        this.nom = nom;
    }

    public String getNom()
    {
        return nom;
    }

    public String parle()
    {
        return "Waf !";
    }

}
```

```
public class Chat implements Animal
{
    private String nom;

    public Chat(String nom)
    {
        this.nom = nom;
    }

    public void setNom(String nom)
    {
        this.nom = nom;
    }

    public String getNom()
    {
        return nom;
    }

    public String parle()
    {
        return "Miaou !";
    }

}

public class Vache implements Animal
{
    private String nom;

    public Vache(String nom)
    {
        this.nom = nom;
    }

    public void setNom(String nom)
    {
        this.nom = nom;
    }

    public String getNom()
    {
        return nom;
    }

    public String parle()
    {
        return "Meuh !";
    }
}
```

Tableau de Comparables

```
import java.util.Random;
import java.lang.Math;

public class ComparableTab
{
    private Comparable[] t;

    /*-------------------------------------------------------*/

    ComparableTab(int taille)
    {
        t = new Comparable[taille];
    }

    /*-------------------------------------------------------*/

    public String toString()
    {
        String res = "[";
        if (t.length >= 1)
            res += t[0];
        for (int i = 1 ; i < t.length ; i++)
            res += ", " + t[i];
        res += "]";
        return res;
    }

    /*-------------------------------------------------------*/
```

```
public Comparable get(int index)
{
    return t[index];
}

/*-------------------------------------------------*/

public void set(int index, Comparable value)
{
    t[index] = value;
}

/*-------------------------------------------------*/

public void echange(int i, int j)
{
    Comparable temp = t[i];
    t[i] = t[j];
    t[j] = temp;
}

/*-------------------------------------------------*/

public void triSelection()
{
    for (int i = 0 ; i < t.length - 1 ; i++)
        {
            int indiceMin = i;
            for (int j = i + 1 ; j < t.length; j++)
                if (t[indiceMin].compareTo(t[j]) > 1)
                    indiceMin = j;
            echange(i, indiceMin);
        }
}

/*-------------------------------------------------*/

public boolean ordonne(int i, int j)
{
    if ((i - j) * (t[i].compareTo(t[j])) < 0)
        {
            echange(i, j);
            return true;
        }
    return false;
}

/*-------------------------------------------------*/

public void triInsertion()
{
    for (int i = 1 ; i < t.length ; i++)
        {
            int j = i;
            while(j > 0 && ordonne(j-1, j))
                j--;
        }
}

/*-------------------------------------------------*/

public void triBulle()
{
    for (int i = t.length - 1 ; i > 0 ; i--)
        for (int j = 0 ; j < i ; j++)
            ordonne(j, j+1);
}

/*-------------------------------------------------*/

public void initTab(ComparableTab other)
{
    t = new Comparable[other.t.length];
    for (int i = 0 ; i < t.length ; i++)
        t[i] = other.t[i];
}

/*-------------------------------------------------*/

public ComparableTab copie()
```

```
    {
        ComparableTab other = new ComparableTab(t.length);
        other.initTab(this);
        return other;
    }

    /*-------------------------------------------------*/

    public void interclasse(ComparableTab t1, ComparableTab t2)
    {
        t = new Comparable[t1.t.length + t2.t.length];
        int i1 = 0, i2 = 0, i = 0;
        while(i < t.length)
            {
                if (i1 == t1.t.length)
                        t[i++] = t2.t[i2++];
                else if (i2 == t2.t.length)
                        t[i++] = t1.t[i1++];
                else if (t1.t[i1].compareTo(t2.t[i2])<0)
                        t[i++] = t1.t[i1++];
                else
                        t[i++] = t2.t[i2++];
            }
    }

    /*-------------------------------------------------*/

    public static void main(String[] args)
    {
        ComparableTab t = new ComparableTab(10);
        Random r = new Random();
        for(int i = 0 ; i < 10 ; i++)
            t.set(i,
                  new RationnelComparable(r.nextInt()%10,
                                          Math.abs(r.nextInt())%10 + 1));
        System.out.println(t);
        t.triInsertion();
        System.out.println(t);
        t.triSelection();
        System.out.println(t);
        t.triBulle();
        System.out.println(t);
        ComparableTab tBis = new ComparableTab(10);
        System.out.println(tBis);
        tBis = t.copie();
        System.out.println(tBis);
        ComparableTab tTer = new ComparableTab(10);
        tBis.triInsertion();
        tTer.interclasse(t, tBis);
        System.out.println(tTer);
        tTer.triInsertion();
        System.out.println(tTer);
    }
}
```

Rationnels comparables

```
public class RationnelComparable implements Comparable
{
    private int num, den;

    /*-------------------------------------------------*/

    public RationnelComparable(int num, int den)
    {
        this.num = num;
        this.den = den;
    }

    /*-------------------------------------------------*/

    public int getNum()
    {
        return num;
    }

    /*-------------------------------------------------*/

    public int getDen()
    {
        return den;
```

```
}

/*-------------------------------------------------*/

public void setNum(int num)
{
    this.num = num;
}

/*-------------------------------------------------*/

public void setDen(int den)
{
    if (den != 0)
        this.den = den;
    else
        System.out.println("Division by Zero !!!");
}

/*-------------------------------------------------*/

public String toString()
{
    return num + "/" + den;
}

/*-------------------------------------------------*/

public RationnelComparable copy()
{
    RationnelComparable r = new RationnelComparable(num, den);
    return r;
}

/*-------------------------------------------------*/

public RationnelComparable opposite()
{
    RationnelComparable r = copy();
    r.setNum(-r.getNum());
    return r;
}

/*-------------------------------------------------*/

private int pgcd(int a, int b)
{
    if (b == 0)
        return a;
    return pgcd(b, a%b);
}

public void reduce()
{
    int p = pgcd(num, den);
    num/=p;
    den*=p;
}

/*-------------------------------------------------*/

public boolean isPositive()
{
    return num > 0 && den > 0 || num < 0 && den < 0;
}

/*-------------------------------------------------*/

public RationnelComparable add(RationnelComparable other)
{
    RationnelComparable res =
        new RationnelComparable(
                            num*other.getDen() + den*other.getNum(),
                            den * other.getDen());
    other.reduce();
    return res;
}

/*-------------------------------------------------*/

public void addBis(RationnelComparable other)
```

```
    {
        num = num*other.getDen() + den*other.getNum();
        den = den * other.getDen() ;
        reduce();
    }

    /*-------------------------------------------------*/

    public RationnelComparable multiply(RationnelComparable other)
    {
        RationnelComparable res = new RationnelComparable(num*other.getNum(),
                                                          den * other.getDen());
        other.reduce();
        return res;
    }

    /*-------------------------------------------------*/

    public RationnelComparable divide(RationnelComparable other)
    {
        RationnelComparable res = new RationnelComparable(num*other.getDen(),
                                                          den * other.getNum());
        other.reduce();
        return res;
    }

    /*-------------------------------------------------*/

    /*
      -1 si this < other
       0 si this = others
       1 si this > other
     */
    public int compareTo(Object other)
    {
        RationnelComparable sub = add(((RationnelComparable)other).opposite());
        if (sub.isPositive())
            return 1;
        if (sub.opposite().isPositive())
            return -1;
        return 0;
    }

    /*-------------------------------------------------*/

    public static void main(String[] args)
    {
        RationnelComparable a, b;
        a = new RationnelComparable(1, 2);
        b = new RationnelComparable(3, 4);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        System.out.println("compareTo(" + a + ", " +
                           b + ") = " + a.compareTo(b));
        System.out.println("compareTo(" + a + ", " +
                           a + ") = " + a.compareTo(a));
        System.out.println("compareTo(" + a + ", " +
                           a.opposite() + ") = " + a.compareTo(a.opposite()));
        System.out.println(a.copy());
        System.out.println(a.opposite());
        System.out.println(a.add(b));
        System.out.println(a.multiply(b));
        System.out.println(a.divide(b));
    }
}
```

3.5.3 Classes abstraites

Animaux

```
public abstract class AnimalAbstrait
{
    private String nom;

    public AnimalAbstrait(String nom)
    {
        setNom(nom);
    }

    public void setNom(String nom)
```

```
    {
        this.nom = nom;
    }

    public String getNom()
    {
        return nom;
    }

    public abstract String parle();
}

public class ChienAbstrait extends AnimalAbstrait
{
    public ChienAbstrait(String nom)
    {
        super(nom);
    }

    public String parle()
    {
        return "Waf !";
    }

}

public class ChatAbstrait extends AnimalAbstrait
{
    private String nom;

    public ChatAbstrait(String nom)
    {
        super(nom);
    }

    public String parle()
    {
        return "Miaou !";
    }

}

public class VacheAbstrait extends AnimalAbstrait
{
    private String nom;

    public VacheAbstrait(String nom)
    {
        super(nom);
    }

    public String parle()
    {
        return "Meuh !";
    }

}
```

Arbres syntaxiques

```
/**
   Fonction d'une variable.
 */

public abstract class Function
{
    /**
       Retourne l'image de x.
     */
    public abstract double evaluate(double x);

    /**
       Retourne la d&eacute;riv&eacute;e.
     */
    public abstract Function derivate();

    /**
       Retourne la fonction simplifi&eacute;e.
     */
    public abstract Function simplify();
```

```
    /**
       Ssi la fonction ne contient pas de variable.
     */
    public abstract boolean isConstant();

    /**
       Ssi la fonction est une feuille valant 0.
     */
    public abstract boolean isZero();

    /**
       Ssi la fonction  est une feuille valant 1.
     */
    public abstract boolean isOne();

    /**
       Retourne l'integrale entre a et b (a < b), calcul&eacute;e avec
       la m&eacute;thode des trap&egrave;zes en
       effectuant nbSubdivisions subdivisions de l'intervalle
       &agrave; int&eacute;grer.
     */
    public double integrate(double a, double b, int nbSubdivisions)
    {
        if (b < a)
            return integrate(b, a, nbSubdivisions);
        double eps = 1./nbSubdivisions;
        double integral = (evaluate(a) + evaluate(b)) / 2;
        for( ; a < b ; a += eps)
            {
                integral += evaluate(a);
            }
        return integral * eps;
    }


    public static void main(String args[])
    {
        Function f = new Minus(new Times(new Constant(4), new Variable()),
                               new Div(new Constant(1), new Variable()));
        System.out.println(f);
        System.out.println(f.evaluate(2));
        System.out.println(f.derivate());
        System.out.println(f.derivate().simplify());
        System.out.println(f.evaluate(20) - f.evaluate(2));
        for (int i = 100000 ; i <= 10000000 ; i+=100000)
            System.out.println("i = " + i + " : "
                               + f.derivate().integrate(2, 20, i));
    }
}

/**
   f(x) = c, o&ugrave; c est une constante r&eacute;elle.
 */

public class Constant extends Function
{
    private double value;

    Constant(double value)
    {
        this.value = value;
    }

    public boolean isZero()
    {
        return value == 0;
    }

    public boolean isOne()
    {
        return value == 1;
    }

    public boolean isConstant()
    {
        return true;
    }

    public Function derivate()
    {
```

```
        return new Constant(0);
    }

    public double evaluate(double x)
    {
        return value;
    }

    public Function simplify()
    {
        return this;
    }

    public String toString()
    {
        return "" + value;
    }
}

/**
   f(x) = x.
 */
public class Variable extends Function
{
    Variable()
    {
    }

    public boolean isZero()
    {
        return false;
    }

    public boolean isOne()
    {
        return false;
    }

    public boolean isConstant()
    {
        return false;
    }

    public Function derivate()
    {
        return new Constant(1);
    }

    public double evaluate(double x)
    {
        return x;
    }

    public Function simplify()
    {
        return this;
    }

    public String toString()
    {
        return "x";
    }
}

/**
   Fonction s'exprimant comme une op&eacute;ration binaire entre
   deux autres fonctions.
 */
public abstract class BinaryOperator extends Function
{
    protected Function leftSon;
    protected Function rightSon;


    BinaryOperator(Function leftSon, Function rightSon)
    {
        this.leftSon = leftSon;
        this.rightSon = rightSon;
    }
```

```
    /**
       Retourne l'op&eacute;rateur binaire sous forme de caract&egrave;re
       ('+' pour une addition '-' pour une soustraction, etc).
     */
    public abstract char toChar();

    public String toString()
    {
        return "(" + leftSon  + " " + toChar()  + " " + rightSon + ")";
    }

    public boolean isZero()
    {
        return false;
    }

    public boolean isOne()
    {
        return false;
    }

    public boolean isConstant()
    {
        return leftSon.isConstant() && rightSon.isConstant();
    }

    /**
       Remplace les sous-arbres par leurs versions simplifi&eacute;es,
       retourne une feuille si l'arbre est constant.
     */
    protected Function simplifySubTrees()
    {
        leftSon = leftSon.simplify();
        rightSon = rightSon.simplify();
        if(isConstant())
            return new Constant(evaluate(0.0));
        return null;
    }

}

/**
   f(x) = g(x) + h(x), o&ugrave; g et h sont les sous-arbres gauche
   et droit.
*/

public class Plus extends BinaryOperator
{
    Plus(Function leftSon, Function rightSon)
    {
        super(leftSon, rightSon);
    }

    public char toChar()
    {
        return '+';
    }

    public double evaluate(double x)
    {
        return leftSon.evaluate(x) + rightSon.evaluate(x);
    }

    public Function derivate()
    {
        return new Plus(leftSon.derivate(), rightSon.derivate());
    }

    public Function simplify()
    {
        Function f = simplifySubTrees();
        if (f != null)
            return f;
        if (leftSon.isZero())
            return rightSon;
        if (rightSon.isZero())
            return leftSon;
        return this;
    }
```

```
}

/**
   f(x) = g(x) - h(x), o&ugrave; g et h sont les sous-arbres gauche
   et droit.
*/

public class Minus extends BinaryOperator
{
    Minus(Function leftSon, Function rightSon)
    {
        super(leftSon, rightSon);
    }

    public char toChar()
    {
        return '-';
    }

    public double evaluate(double x)
    {
        return leftSon.evaluate(x) - rightSon.evaluate(x);
    }

    public Function derivate()
    {
        return new Minus(leftSon.derivate(), rightSon.derivate());
    }

    public Function simplify()
    {
        Function f = simplifySubTrees();
        if (f != null)
            return f;
        if (rightSon.isZero())
            return leftSon;
        if (leftSon.isZero())
            return new Times(new Constant(-1), rightSon);
        return this;
    }
}

/**
   f(x) = g(x) * h(x), o&ugrave; g et h sont les sous-arbres gauche
   et droit.
*/

public class Times extends BinaryOperator
{
    Times(Function leftSon, Function rightSon)
    {
        super(leftSon, rightSon);
    }

    public char toChar()
    {
        return '*';
    }

    public double evaluate(double x)
    {
        return leftSon.evaluate(x) * rightSon.evaluate(x);
    }

    public Function derivate()
    {
        return new Plus(new Times(leftSon.derivate(), rightSon),
                        new Times(leftSon, rightSon.derivate()));
    }

    public Function simplify()
    {
        Function f = simplifySubTrees();
        if (f != null)
            return f;
        if (rightSon.isZero() || leftSon.isZero())
            return new Constant(0);
        if (rightSon.isOne())
            return leftSon;
        if (leftSon.isOne())
            return rightSon;
```

```
        return this;
    }

}

/**
   f(x) = g(x) / h(x), o&ugrave; g et h sont les sous-arbres gauche
   et droit.
*/

public class Div extends BinaryOperator
{
    Div(Function leftSon, Function rightSon)
    {
        super(leftSon, rightSon);
    }

    public char toChar()
    {
        return '/';
    }

    public double evaluate(double x)
    {
        return leftSon.evaluate(x) / rightSon.evaluate(x);
    }

    public Function derivate()
    {
        return new Div(
                        new Minus(new Times(leftSon.derivate(), rightSon),
                                  new Times(leftSon, rightSon.derivate())),
                        new Times(rightSon, rightSon)
                        );
    }

    public Function simplify()
    {
        Function f = simplifySubTrees();
        if (f != null)
            return f;
        if (leftSon.isZero())
            return new Constant(0);
        if (rightSon.isOne())
            return leftSon;
        if (leftSon.isOne())
            return rightSon;
        return this;
    }
}
```

3.6 Exceptions

Classe inutile

```
public class ClasseInutile
{
    public void nePasInvoquer() throws ExceptionInutile
    {
        throw new ExceptionInutile();
    }

    public static void main(String[] args)
    {
        ClasseInutile o = new ClasseInutile();
        try
            {
                o.nePasInvoquer();
            }
        catch(ExceptionInutile e)
            {
                System.out.println(e);
            }
    }
}

class ExceptionInutile extends Exception
{
    public ExceptionInutile()
```

```
    {
        System.out.println("Je vous avais dit de ne pas " +
                           "invoquer cette fonction !");
    }

    public String toString()
    {
        return "Vous avez essayer d'invoquer une methode " +
            "qu'il ne fallait pas invoquer !";
    }
}
```

Pile

```
public class ObjectStack
{
    /*
        Liste contenant les elements de la pile.
    */

    private ObjectList l;

    /*****************************************/

    /*
        Taille de la pile
    */

    private final int taille;

    /*****************************************/

    /*
        Nombre d'elements dans la liste
    */

    private int nbItems;

    /*****************************************/

    /*
        Constructeur
    */

    ObjectStack(int taille)
    {
        l = null;
        this.taille = taille;
        nbItems = 0;
    }

    /*****************************************/

    /*
        Retourne vrai si et seulement
        si la pile est vide
    */

    public boolean estVide()
    {
        return nbItems == 0;
    }

    /*****************************************/

    /*
        Retourne vrai si et seulement si la pile est
        pleine.
    */

    public boolean estPleine()
    {
        return nbItems == taille;
    }

    /*****************************************/

    /*
        Retourne l'element se trouvant au sommet de
        la pile, -1 si la pile est vide.
    */
```

```
    public Object sommet()
    {
        if (!estVide())
            return l.getData();
        return -1;
    }

    /*****************************************/

    /*
        Supprime l'element se trouvant au sommet
        de la pile, ne fait rien si la pile est
        vide.
    */

    public void depile() throws PileVideException
    {
        if (!estVide())
            {
                l = l.getNext();
                nbItems--;
            }
        else
            {
                throw new PileVideException();
            }
    }

    /*****************************************/

    /*
        Ajoute data en haut de la pile, ne fait rien
        si la pile est pleine.
    */

    public void empile(int data)
    {
        l = new ObjectList(data, l);
        nbItems ++;
    }

    /*****************************************/

    /*
        Retourne une representation de la pile
        au format chaine de caracteres.
    */

    public String toString()
    {
        return l.toString();
    }

    /*****************************************/

    /*
        Teste le fonctionnement de la pile.
    */

    public static void main(String[] args)
    {
        ObjectStack p = new ObjectStack (30);
        int i = 0;
        while(!p.estPleine())
            p.empile(new Integer(i++));
        System.out.println(p);
        while(!p.estVide())
            {
                System.out.println(p.sommet());
                try
                    {
                    p.depile();
                    }
                catch(PileVideException e)
                    {
                        System.out.println(e);
                    }
            }
    }
}
```

3.7 Interfaces graphiques

3.7.1 Écouteurs d'événement

```
import javax.swing.*;
import java.awt.event.*;

public class FormaterDisqueDur extends JFrame implements ActionListener
{
    public FormaterDisqueDur()
    {
        setTitle("Gestionnaire du disque dur");
        setSize(100, 100);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JButton formater = new JButton("Formater le disque dur");
        getContentPane().add(formater);
        formater.addActionListener(this);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e)
    {
        System.out.println("Formatage en cours.");
    }

    public static void main(String[] args)
    {
        FormaterDisqueDur f = new FormaterDisqueDur();
    }
}

import javax.swing.*;
import java.awt.event.*;

public class FormaterDisqueDurAnonyme extends JFrame
{


    public FormaterDisqueDurAnonyme()
    {
        setTitle("Gestionnaire du disque dur");
        setSize(100, 100);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JButton formater = new JButton("Formater le disque dur");
        getContentPane().add(formater);
        formater.addActionListener(new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    System.out.println("Formatage en cours.");
                }
            }
            );
        setVisible(true);
    }

    public static void main(String[] args)
    {
        FormaterDisqueDurAnonyme f = new FormaterDisqueDurAnonyme();
    }
}

import javax.swing.*;
import java.awt.event.*;

public class FormaterDisqueDurNonAnonyme extends JFrame
{
    public FormaterDisqueDurNonAnonyme()
    {
        setTitle("Gestionnaire du disque dur");
        setSize(100, 100);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JButton formater = new JButton("Formater le disque dur");
        getContentPane().add(formater);
        formater.addActionListener(new AffichageFormatage());
        setVisible(true);
    }

    public static void main(String[] args)
```

```
    {
        FormaterDisqueDurNonAnonyme f = new FormaterDisqueDurNonAnonyme();
    }
}

class AffichageFormatage implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.out.println("Formatage en cours.");
    }
}
```

3.7.2 Gestionnaires de mise en forme

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class FormaterDisqueDurGridLayout extends JFrame
{
    public FormaterDisqueDurGridLayout()
    {
        setTitle("Gestionnaire du disque dur");
        setSize(100, 100);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        JButton formater = new JButton("Formater le disque dur");
        final JLabel label = new JLabel("");
        formater.addActionListener(new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    label.setText("Formatage en cours.");
                }
            }
            );
        getContentPane().setLayout(new GridLayout(2, 1));
        getContentPane().add(formater);
        getContentPane().add(label);
        setVisible(true);
    }

    public static void main(String[] args)
    {
        FormaterDisqueDurGridLayout f = new FormaterDisqueDurGridLayout();
    }
}
```

Convertisseur Euro/Dollar

```
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class EuroDollar extends JFrame
{
    final double OneDollarInEuro = 1.4237 ;
    final JTextField dollarText, euroText;
    final JLabel dollarLabel, euroLabel;

    private void convert(JTextField source, JTextField target, double rate)
    {
        try
            {
                double k = new Double(source.getText()).doubleValue();
                k *= rate;
                target.setText((new Double(k)).toString());
            }
        catch(NumberFormatException ex)
            {
                if(target != null)
                    target.setText("");
            }
    }

    public EuroDollar()
    {
        getContentPane().setLayout(new GridLayout(2, 2));
        dollarText = new JTextField();
        euroText = new JTextField();
```

```
        dollarLabel = new JLabel("Dollars");
        euroLabel = new JLabel("Euros");
        getContentPane().add(dollarLabel);
        getContentPane().add(euroLabel);
        getContentPane().add(dollarText);
        getContentPane().add(euroText);
        setTitle("Convertisseur Euros/Dollars");
        setSize(400, 50);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
        euroText.addKeyListener(new KeyListener()
            {
                public void keyTyped(KeyEvent e){}
                public void keyPressed(KeyEvent e){}
                public void keyReleased(KeyEvent e)
                {
                    convert(euroText, dollarText, OneDollarInEuro);
                }
            }
            );
        dollarText.addKeyListener(new KeyListener()
            {
                public void keyTyped(KeyEvent e){}
                public void keyPressed(KeyEvent e){}
                public void keyReleased(KeyEvent e)
                {
                    convert(dollarText, euroText, 1/OneDollarInEuro);
                }
            }
            );
    }

    public static void main(String[] args)
    {
        EuroDollar maFenetre = new EuroDollar();
    }
}
```

3.8 Collections

3.8.1 Deux objets

```
public class DeuxObjets<T extends Comparable<T>>
    implements Comparable<DeuxObjets<T>>
{
    private T first;
    private T second;

    public DeuxObjets(T first, T second)
        {
            this.first = first;
            this.second = second;
        }

    public T getPetit()
    {
        if (first.compareTo(second) < 0)
            return first;
        else
            return second;
    }

    public T getGrand()
    {
        if (first.compareTo(second) >= 0)
            return first;
        else
            return second;
    }

    public int compareTo(DeuxObjets<T> other)
    {
        return  getGrand().compareTo(other.getGrand());
    }

    public String toString()
    {
        return "(" + first + ", " + second + ")";
    }
```

```
    public static void main(String args[])
    {
        DeuxObjets<Integer> a = new DeuxObjets<Integer>(2, 7);
        System.out.println(a.getGrand());
    }
}
```

3.8.2 Package Pile

Stack.java

```
package alexandre;

import java.util.Iterator;

public class Stack<T> implements Iterable<T>
{
    /*
        Liste contenant les elements de la pile.
    */

    private List<T> l;

    /*****************************************/

    /*
        Constructeur
    */

    Stack()
    {
        l = null;
    }

    /*****************************************/

    /*
        Retourne vrai si et seulement
        si la pile est vide
    */

    public boolean estVide()
    {
        return l == null;
    }

    /*****************************************/

    /*
        Retourne l'element se trouvant au sommet de
        la pile.
    */

    public T sommet() throws PileVideException
    {
        if (!estVide())
            return l.getData();
        throw new PileVideException();
    }

    /*****************************************/

    /*
        Supprime l'element se trouvant au sommet
        de la pile, ne fait rien si la pile est
        vide.
    */

    public void depile()
    {
        if (!estVide())
            l = l.getNext();
    }

    /*****************************************/

    /*
        Ajoute data en haut de la pile.
    */
```

```
    public void empile(T data)
    {
        l = new List<T>(data, l);
    }

    /*********************************/

    /*
      Retourne un iterateur sur la pile.
     */

    public Iterator<T> iterator()
    {
        return l.iterator();
    }

    /*****************************************/

    /*
        Retourne une representation de la pile
        au format chaine de caracteres.
    */

    public String toString()
    {
        String res = "";
        for(T m : this)
            res += m + " ";
        return res;
    }

    /*****************************************/

    /*
        Teste le fonctionnement de la pile.
    */

    public static void main(String[] args)
    {
        Stack<Integer> p = new Stack<Integer> ();
        int i = 0;
        while(i < 20)
            p.empile(i++);
        System.out.println(p);
        while(!p.estVide())
            {
                try
                    {
                        System.out.println(p.sommet());
                    }
                catch(PileVideException e)
                    {
                        System.out.println(e);
                    }
                p.depile();
            }
    }
}
```

List.java

```
package alexandre;

import java.util.Iterator;

class List<T> implements Iterable<T>
{
    /*
      Donnee stockee dans le maillon
    */

    private T data;

    /*********************************/

    /*
      Pointeur vers le maillon suivant
     */

    private List<T> next;
```

```
/*********************************/

/*
  Constructeur initialisant la donnee et
  le pointeur vers l'element suivant.
 */

List(T data, List<T> next)
{
    this.data = data;
    this.next = next;
}

/*********************************/

/*
  Constructeur initialisant la donnee et
  mettant le suivant a null.
 */

List(T data)
{
    this(data, null);
}

/*********************************/

/*
  Retourne la donnee.
 */

public T getData()
{
    return data;
}
/*********************************/

/*
  Modifie la donnee
 */

public void setData(T data)
{
     this.data = data;
}
/*********************************/

/*
  Retourne le maillon suivant.
 */

public List<T> getNext()
{
    return next;
}

/*********************************/

/*
  Modifie le maillon suivant
 */

public void setNext(List<T> next)
{
    this.next = next;
}

/*********************************/

/*
  Retourne un iterateur sur la liste.
 */

public Iterator<T> iterator()
{
    return new MyIterator<T>(this);
}

/*********************************/
```

```
    /*
      Retourne une reprÃ©sentation sous forme de
      chaine de la liste.
     */

    public String toString()
    {
        String res = "" + data;
        if (next != null)
            res += " -> " + next.toString();
        return res;
    }

    /*********************************/

    /*
      Teste le fonctionnement de la liste.
     */

    public static void main(String[] args)
    {
        List<Integer> l = new List<Integer>(20);
        int i = 19;
        while (i >= 0)
            l = new List<Integer>(i--, l);
        System.out.println(l);
    }

}
```

PileVideException.java

```
package alexandre;

public class PileVideException extends Exception
{
    public String toString()
    {
        return "Can't top an empty stack.";
    }
}
```

MyIterator.java

```
package alexandre;

import java.util.Iterator;

public class MyIterator<T> implements Iterator<T>
{
    private List<T> l;

    MyIterator(List<T> l)
        {
            this.l = l;
        }

    public boolean hasNext()
    {
        return  l != null;
    }

    public T next()
    {
        List<T> temp = l;
        l = l.getNext();
        return temp.getData();
    }

    public void remove()
    {
    }
}
```

3.8.3 Parcours

Parcours de ArrayList

```
import java.util.ArrayList;
import java.util.Random;

public class ParcoursArrayList
{
    public static void main(String[] args)
    {
        ArrayList<Integer> a = new ArrayList<Integer>();
        Random r = new Random();
        for (int i = 1 ; i <= 40 ; i++)
            a.add(r.nextInt());
        for(int i : a)
            System.out.println(i);
    }
}
```

Parcours de `TreeSet`

```
import java.util.TreeSet;
import java.util.Random;

public class ParcoursTreeSet
{
    public static void main(String[] args)
    {
        TreeSet<DeuxObjets<Integer>> a = new TreeSet<DeuxObjets<Integer>>();
        Random r = new Random();
        for (int i = 1 ; i <= 40 ; i++)
            a.add(new DeuxObjets<Integer>(r.nextInt(), r.nextInt()));
        for(DeuxObjets<Integer> i : a)
            System.out.println(i);
    }
}
```

3.9 Threads

3.9.1 Prise en main

```
public class BinaireAleatoireRunnable implements Runnable
{
    private int value;
    private int nbIterations;

    public BinaireAleatoireRunnable(int value, int nbIterations)
    {
        this.value = value;
        this.nbIterations = nbIterations;
    }

    public void run()
    {
        for (int i = 1 ; i <= nbIterations ; i++)
            System.out.print(value);
    }

    public static void main(String[] args)
    {
        Runnable un = new BinaireAleatoireRunnable(1, 30000);
        Runnable zero = new BinaireAleatoireRunnable(0, 30000);
        Thread tUn = new Thread(un);
        Thread tZero = new Thread(zero);
        tUn.start();
        tZero.start();
    }
}
```

3.9.2 Synchronisation

Compteur partagé (méthode synchronisée)

```
public class BinaireSynchronise extends Thread
{
    private int value;
    private Counter c;

    public BinaireSynchronise(int value, Counter c)
    {
```

```
        this.value = value;
        this.c = c;
    }

    public void run()
    {
        while (c.stepCounter())
            System.out.print(value);
    }

    public static void main(String[] args)
    {
        Counter c = new Counter(30000);
        Thread un = new BinaireSynchronise(1, c);
        Thread zero = new BinaireSynchronise(0, c);
        un.start();
        zero.start();
    }
}

class Counter
{
    private int i;
    private final int nbIterations;

    Counter(int nbIterations)
    {
        this.nbIterations = nbIterations;
        i = 1;
    }

    synchronized boolean stepCounter()
    {
        if (i > nbIterations)
            return false;
        i++;
        return true;
    }
}
```

Compteur partagé (section critique)

```
public class BinaireSynchroniseBis extends Thread
{
    private int value;
    private Counter c;

    public BinaireSynchroniseBis(int value, Counter c)
    {
        this.value = value;
        this.c = c;
    }

    public void run()
    {
        boolean okay = true;
        while (okay)
            synchronized(c)
                {
                    okay = c.stepCounter();
                    if (okay)
                        System.out.print(value);
                }
    }

    public static void main(String[] args)
    {
        Counter c = new Counter(30000);
        Thread un = new BinaireSynchroniseBis(1, c);
        Thread zero = new BinaireSynchroniseBis(0, c);
        un.start();
        zero.start();
    }
}

class Counter
{
    private int i;
    private final int nbIterations;

    Counter(int nbIterations)
```

```
    {
        this.nbIterations = nbIterations;
        i = 1;
    }

    boolean stepCounter()
    {
        if (i > nbIterations)
            return false;
        i++;
        return true;
    }
}
```

3.9.3 Mise en attente

```
import java.util.Random;

public class ProducteurConsommateur
{
    public static void main(String[] args)
    {
        File<Integer> f = new File<Integer>(30);
        Thread producteur1 = new Producteur(f, "producteur 1");
        Thread producteur2 = new Producteur(f, "producteur 2");
        Thread consommateur1 = new Consommateur(f, "consommateur 1");
        Thread consommateur2 = new Consommateur(f, "consommateur 2");
        producteur1.start();
        producteur2.start();
        consommateur1.start();
        consommateur2.start();
    }
}

class Producteur extends Thread
{
    private File<Integer> file;
    String name;

    Producteur(File<Integer> file, String name)
    {
        this.file = file;
        this.name = name;
    }

    public void run()
    {
        Random r = new Random();
        while(true)
            {
                synchronized(file)
                    {
                        try
                            {
                                sleep(100);
                            }
                        catch(InterruptedException e)
                            {
                                System.out.println(e);
                            }
                        System.out.println(name);
                        if (!file.isFull())
                            {
                                boolean empty = file.isEmpty();
                                try
                                    {
                                        file.add(r.nextInt());
                                    }
                                catch(Exception e)
                                    {
                                        System.out.println(e);
                                    }
                                if (empty)
                                    file.notifyAll();
                            }
                        else
                            try
                                {
                                    file.wait();
                                }
                            catch(InterruptedException e)
```

```
                                    {
                                        System.out.println(e);
                                    }
                        }
                }
        }
}

class Consommateur extends Thread
{
    private File<Integer> file;
    private String name;

    Consommateur(File<Integer> file, String name)
    {
        this.file = file;
        this.name = name;
    }

    public void run()
    {
        while(true)
            {
                synchronized(file)
                    {
                        try
                            {
                                sleep(100);
                            }
                        catch(InterruptedException e)
                            {
                                System.out.println(e);
                            }
                        System.out.println(name);
                        if (!file.isEmpty())
                            {
                                boolean full = file.isFull();
                                int first = 0;
                                try
                                    {
                                        first = file.getFirst();
                                        file.removeFirst();
                                    }
                                catch(Exception e)
                                    {
                                        System.out.println(e);
                                    }
                                System.out.println(first);
                                if (full)
                                    file.notifyAll();
                            }
                        else
                            try
                                {
                                    file.wait();
                                }
                            catch(InterruptedException e)
                                {
                                    System.out.println(e);
                                }
                    }
            }
    }
}

class List<T>
{
    private T data;
    private List<T> next;

    List(T data, List<T> next)
        {
            this.data = data;
            this.next = next;
        }

    List(T data)
        {
            this(data, null);
        }
```

```
    public T getData()
    {
        return data;
    }

    public List<T> getNext()
    {
        return next;
    }

    public void setNext(List<T> next)
    {
        this.next = next;
    }
}

class EmptyFileException extends Exception
{
    public EmptyFileException()
    {
        System.out.println("Can't read empty file");
    }
}

class FullFileException extends Exception
{
    public FullFileException()
    {
        System.out.println("Can't write full file");
    }
}

class File<T>
{
    private List<T> first = null;
    private List<T> last = null;
    private int nbElements = 0;
    private final int maxNbElements;


    public File(int maxNbElements)
        {
            this.maxNbElements = maxNbElements;
        }

    public T getFirst() throws EmptyFileException
    {
        return first.getData();
    }

    public void removeFirst() throws EmptyFileException
    {
        first = first.getNext();
        nbElements--;
    }

    public void add(T data) throws FullFileException
    {
        if (isEmpty())
            {
                first = last = new List<T>(data);
            }
        else
            {
                last.setNext(new List<T>(data));
                last = last.getNext();
            }
        nbElements++;
    }

    public boolean isEmpty()
    {
        return nbElements == 0;
    }

    public boolean isFull()
    {
        return nbElements == maxNbElements;
    }
}
```

```
import java.util.Random;

public class Philosophes extends Thread
{
    private final int nbPlaces = 5;
    private Couvert[] couverts;
    private Place[] places;

    public Philosophes()
    {
        couverts = new Couvert[nbPlaces];
        for (int i = 0 ; i < nbPlaces ; i++)
            couverts[i] = new Couvert(i);
        places = new Place[nbPlaces];
        for (int i = 0 ; i < nbPlaces ; i++)
            places[i] = new Place(i, couverts[i],
                                  couverts[(i + 1)%nbPlaces]);
    }

    public void run()
    {
        Random r = new Random();
        int i = 0;
        while(true)
            {
                Philosophe p = new Philosophe(i++,
                                              places[r.nextInt(nbPlaces)]);
                p.start();
                try {
                    sleep(500);}
                catch(InterruptedException e){}
            }
    }

    public static void main(String[] args)
    {
        Philosophes p = new Philosophes();
        p.start();
    }
}

class Place
{
    private Couvert couvertGauche;
    private Couvert couvertDroit;
    private Philosophe philosophe;
    private int indice;

    Place(int indice, Couvert couvertGauche, Couvert couvertDroit)
    {
        this.indice = indice;
        this.couvertGauche = couvertGauche;
        this.couvertDroit = couvertDroit;
    }

    void occuper(Philosophe philosophe)
    {
        this.philosophe = philosophe;
        System.out.println(philosophe + "a pris la " + this);
    }

    void liberer()
    {
        System.out.println(philosophe + "a libere la " + this);
        philosophe = null;
    }

    void manger()
    {
        synchronized(couvertGauche)
            {
                couvertGauche.prend(this);
                synchronized(couvertDroit)
                    {
                        couvertDroit.prend(this);
                        System.out.println(philosophe +
                                           " commence a manger a " + this);
                        try{Thread.sleep(5000);}
                        catch(Exception e){}
                        System.out.println(philosophe +
                                           " a fini de manger a " + this);
```

```
                        couvertDroit.repose(this);
                    }
                couvertGauche.repose(this);
            }
    }

    public String toString()
    {
        return "place " + indice + " ";
    }
}

class Couvert
{
    private int indice;

    public Couvert(int indice)
    {
        this.indice = indice;
    }

    void prend(Place place)
    {
        System.out.println(this + "pris par " + place);
    }

    void repose(Place place)
    {
        System.out.println(this + "repose par " + place);
    }

    public String toString()
    {
        return "couvert " + indice + " ";
    }
}

class Philosophe extends Thread
{
    private Place place;
    private int indice;

    Philosophe(int indice, Place place)
    {
        this.indice = indice;
        this.place = place;
    }

    public void run()
    {
        System.out.println(this + "en attente de " + place);
        synchronized(place)
            {
                place.occuper(this);
                place.manger();
                place.liberer();
            }
    }

    public String toString()
    {
        return "philosophe " + indice + " ";
    }

}
```