POLYCOPIE SUR LE LANGAGE JAVA

Sommaire

CHAPITRE 2: DECOUVERTE DU LANGAGE 6
public class Primitive 9
Les méthodes *final* ne peuvent pas être redéfinies dans les classes dérivées. 9
Un attribut sans modificateur (modificateur par défaut) n'est pas accessible depuis l'extérieur du paquet de la classe. 12
La taille d'un tableau est fixe. Néanmoins, il n'est pas nécessaire qu'elle soit connue au moment de la compilation. 12
Exemple 12
public class chaines 13
Bonjour 14
Les chaînes de caractères peuvent être concaténées 14
String chaine1 = "Bonjour"; 14
• Supposons qu'on dispose de 2 classe A et B, si B hérite de A, on appelle B la sous classe et A la super classe. 15
Pour définir une sous-classe, on utilise le mot clé *extends* 15
Exemple 15
Exemple 16
Class Canari extends Animal 16
2 Constructeur de sous-classe 16
Remarque 16
Exemple 18
5 Les interfaces 18
Exemple 19
Exemple 19
Public interface UneInterface 19
Remarque 19
Exemple 20
public class Ville 20
2 Mécanisme des exceptions 20
Déclaration 20
3 Propager ou capturer une exception 21
Supposons que dans la classe Ville, on a un constructeur avec 2 arguments qui appelle la méthode SetNbHabitants( ); 21
Exemple 21
Propager 21
On utilise le *throws* 21
Capturer 21
Exemple 21
CHAPITRE 5 : LES ENTREES / SORTIES 22
1 Généralités sur les flots de données 22
Flots d'octets 22
Flots de caractères 22
Exemple 22
2 Saisir des données envoyées par le clavier 22
Remarques!! 23
Exemple complet 23
3 Lire ou écrire des caractères dans un fichier 23
Exemple 24
Remarque 24
4 Ecrire et lire dans un fichier texte 24
Exemple 24
Remarque 25
Exemple 25
CHAPITRE VI: INTERFACES GRAPHIQUES 26
1 Introduction 26
Remarque 31
La classe URL 31
Lecture et Ecriture dans une URL en utilisant URLConnection 32
Exemple 33

CHAPITRE 1 : GENERALITES

1 Qu’est-ce que Java ?

Le langage Java a été introduit par la société SUN en 1995. Il possède de nombreuses caractéristiques :

- C’est un langage orienté objet
- C’est un langage compilé : avant d’être exécuté, il doit être traduit dans le langage de la machine sur laquelle il doit fonctionner
- Il emprunte sa syntaxe en grande partie du langage C
- Les programme Java peuvent être exécutés sous forme d’applications indépendantes ou distribuées à travers le réseau et exécutées par un navigateur Internet sous forme d’applets.

2 Pourquoi utiliser Java ?

2.1 Le monde sans Java

Avec les langages évolués courant (C++, C, etc.) nous avons pris l’habitude de coder sur une machine identique à celle qui exécutera nos applications ; la raison est fort simple : à de rares exceptions prés les compilateurs ne sont pas multi-plateformes et le code généré est spécifique à la machine qui doit accueillir. Nous devons alors utiliser *n* compilateurs différents sur *n* machines. Aujourd’hui, la généralisation des interfaces graphiques et l’usage de langage plus évolués compliquent encore d’avantage le problème. Ainsi pour développer une application destinées à plusieurs systèmes d’exploitation avec ses différentes couches de librairies et d’interfaces ; les API de ces interfaces étant toutes différentes. Ainsi nos applications sont fortement dépendantes des ressources (y compris graphique) du système hôte, dépendantes des API des interfaces utilisées, et le code produit ne peut s’exécuter que sur le système pour lequel il a été initialement produit.

2.2 Le monde avec Java

Tout d’abord, Java simplifie le processus de développement : quelle que soit la machine sur laquelle on code, le compilateur fournit le même code. Ensuite, quel que soit le système utilisé cet unique code est directement opérationnel: En effet, la compilation d’un source Java produit du pseudo-code (byte code) Java qui sera exécuté par tout interpréteur Java sans aucune modification ou recompilation. Cet « interpréteur » est couramment dénommé « machine virtuelle Java ».

3 Utilisation du JDK (Kit de développement Java)

Permet le développement de programmes en Java. Il est constitué de plusieurs outils tel que :

- javac.exe : compilateur
- java.exe : interpréteur
- jdb.exe : debugger…

et d’une importante librairies de classe (API).

Téléchargement :

http://www.sun.com/products/index.html

Configuration :

Chemin d'accès aux exécutables

En supposant que vous avez choisi la version *windows 98*. Vous devez modifier la variable *PATH* du fichier *autoexec.bat* :

- Ouvrez à l'aide d'un éditeur de texte, le fichier *autoexec.bat* se trouvant dans la racine du disque dur
- Localisez la ligne commençant par set *path* et ajoutez à la fin de celle-ci : set path = c:\jdk1.2\bin

La variable d'environnement 'path' indique à Windows le chemin d'accès qu'il doit utiliser pour trouver les programmes exécutables. Windows cherche les programmes exécutables tout d'abord dans le dossier à partir duquel la commande est tapée, puis dans les dossiers dont les chemins d'accès sont indiqués par la variable « path ».

Chemin d'accès aux classes Java

Le chemin d'accès aux classes Java peut être configuré exactement de la même façon à l'aide de la variable *classpath.*

4 Syntaxe du langage Java

4.1 Type de variables

En Java on dispose des mêmes types qu'en langage C (int, float, char…). On dispose d'en plus du type *boolean (1 bit).* Ce type a deux valeurs possibles *false* et *true* (initialisation à false).

4.2 Opérateurs

Opérateurs arithmétiques

+ - * / % ++ --

Opérateurs d'affectation

= += -= *= /=

Opérateurs de comparaison

< > <= >=

Opérateurs logiques

! && ||

4.3 Structures de contrôle et débranchements

Boucles répétitives

while do…while for

Instructions pour faire des choix

if…else switch…case

5 Principe de la programmation en Java : l'orienté objet

Un programme structuré (Pascal, C…) est composé de fonctions indépendantes constituées d'instructions simples et structurés. Ainsi, les données et les fonctions qui opèrent sur elles sont séparées. Les données apparaissent généralement en tête du programme et sont donc visibles de toutes les fonctions qui suivent. Cette organisation pose le grave problème des effets de bord. En entrant dans une fonction, on est jamais assuré de trouver les données dans l'état attendu, car n'importe quelle autre fonction, même si ce n'est pas son rôle, peut les modifier. De plus, à cause de cette séparation entre données et fonctions, de profonds bouleversements du programme sont nécessaires quand les structures de données sont modifiées. La solution à ces problèmes est la programmation objet. Elle est basée sur trois principes fondamentaux : l'encapsulation, l'héritage, et le polymorphisme.

5.1 L'encapsulation

L'encapsulation des données est le premier et le plus important des concepts de la programmation objet. Il stipule que les données et les fonctions qui opèrent sur elles sont encapsulées dans des objets. Les seules fonctions a être autorisées à modifier les données d'un objet sont les fonctions appartenant à cet objet. Depuis l'extérieur d'un objet, on ne peut le modifier que par des fonctions faisant office d'interface. Ainsi, il n'est plus à craindre que des fonctions modifient indûment des données.

5.2 L'héritage

Les objets, comme les données, possèdent un type que l'on appelle classe. Les classes peuvent être organisées en hiérarchies, chaque classe héritant de sa classe mère ou super-classe. L'héritage est ainsi source d'économie de code, une classe peut être définie comme la descendante d'une classe (ou sous-classe).

5.3 Le polymorphisme

Le troisième principe de base de la programmation objet est le polymorphisme. Il passe plus inaperçu que les précédentS. Des fonctions différentes dans les classes différentes peuvent prendre le même nom. Ainsi, dans une hiérarchie de classes d'éléments graphiques la fonction *dessiner( )* aura le même nom pour un polygone ou un cercle, cependant les techniques utilisées pour dessiner ces éléments sont différentes. Le polymorphisme est beaucoup plus puissant qu'il n'y paraît à première vue. Il fait économiser des identificateurs de fonctions et rend les notations plus lisibles.

CHAPITRE 2: DECOUVERTE DU LANGAGE

1 Définition d'une classe et d'un objet en Java

La notion de classe est un enrichissement de la notion usuelle de structure en C *(struct)* en C. Une classe permet de définir un type d'objet, éventuellement compliqué, associant des données et des procédures qui accèdent à ces données. Les données se présentent sous forme de champ désignés par identificateurs et dotés d'un type. A une donnée, on associe un mot clé qui désigne l'accessibilité de cette donnée.
Les procédures, également appelées *méthodes*, définissent les opérations possibles sur ces composants. A une méthode, on associe un mot clé désignant l'accessibilité de cette méthode.

Exemple

```
class Point {

        private int _x;
        private int _y;

        public Point()
        {
          _x = 0 ; _y = 0 ;
        }

        public Point(int x, int y)
        {
          _x = x ; _y = y ;
        }

        public void avancer(int dx, int dy)
        {
          _x = _x + dx ; _y = _y + dy ;
        }

    }
```

La classe Point possède deux attribut : *_x*, *_y* et trois méthodes *Point()*, *Point(int, int)*, *avancer(int, int).*

Le mot clé ***private*** indique que l'attribut est privé il ne sera accessible que par les méthodes appartenant à la classe. Un attribut **public** est accessible par une méthode appartenant à une autre classe (voir aussi paragraphe 9).

Remarque

Le corps (contenu) des méthodes d'une classe sera défini dans la classe elle-même.

Initialisation des objets : constructeur

Une classe peut définir une ou plusieurs procédures d'initialisation appelées *constructeurs*. Un constructeur est une méthode avec aucune valeur de retour et qui porte le même nom que la classe. S'il existe plusieurs méthodes dans une classe, elles se distinguent par le type ou le nombre de paramètres :

Point( ) et *Point(int, int)*

Le constructeur permet de créer une instance de la classe. Une instance de classe est dite *objet*.

Pour créer un Point on va :

1. Déclarer un *handle* sur un Point.

Point p ; (p est un handle sur un point).

Remarque

Un handle est un numéro. Par exemple, quand une application alloue de la mémoire, le compilateur ne rend pas un pointeur mais un numéro. Ce numéro est associé à une zone mémoire.

2. La déclaration seule ne crée pas d'objet. Elle associe l'identificateur à une référence appelée *null* qui ne fait référence à rien.

p | null |

Pour créer une instance de la classe point (on dit objet de la classe Point), il faut exécuter **new** en citant un constructeur de la classe Point, avec ses paramètres. La primitive **new** rend en résultat la référence sur l'objet crée, qu'il suffit alors d'affecter à la variable déclarée :

new Point(2,0)

Point p = new Point(2,0) ;

destruction des objets: destructeur

Avec Java, plus besoin d'écrire une fonction destructeur. En effet, il existe un programme appelé 'garbage collector' (ramasseur d'ordures) qui est exécuté automatiquement dés que la mémoire disponible devient inférieure à un certain seuil.

2 Compilation et exécution d’un programme en java

Pour compiler un programme en Java, il faut utiliser l’instruction *javac* suivi du nom du fichier.

Exemple

Pour compiler la classe Point, on écrit javac Point.java

A la fin de la compilation si aucune erreur n’a été détectée, le code de base est transformé en byte code et un fichier *Point.class* est généré. C’est ce fichier qui sera exécuté.

Pour exécuter un programme en Java, on utilise l’instruction java suivi du nom de la classe.

Exemple

Pour exécuter la classe Point, on écrit java Point

Remarque

Quand on écrit un programme en Java il faut éviter de donner au fichier .java un nom différent de la classe. Par exemple, si pour on sauvegarder la classe Point nous choisissons le nom Toto.java à la fin de la compilation le fichier Toto.class sera créer. Il y aura ensuite des erreurs au niveau de l’exécution (en effet, la classe Toto n’existe pas).

3 Notion de primitives

Souvent on dit que le langage Java est un langage tout objet (tout est instance d'une classe). En fait, ce n'est pas tout a fait vrai. Supposons qu'on aune boucle permettant d'incrémenter u entier 1000 fois, il ne serait pas judicieux de créer 1000 un objet pour l'utiliser comme incrément (il existe en Java la classe *Integer* qui pourrait être utilisée mais qui ralentirai énormément le programme avec une création de 1000 objets *Integer*). Les concepteurs de Java ont donc doté ce langage d'une série d'éléments particuliers appelés *primitives* qui ne sont pas gérés de la même manière en mémoire (voir chapitre 1 du polycopie pour les différentes primitives existantes).

Différence entre les objets et les primitives

Les primitives peuvent être "enveloppées" dans un objet provenant d'une classe prévue à cet effet et appelée Wrapper (mot anglais signifiant envelopppeur). Les enveloppeurs sont donc des objets pouvant contenir une primitive et auxquels sont associés des méthodes permettant de les manipuler.

4 Affichage sur sortie standard (écran)

Pour afficher un commentaire sur écran, on utilise la méthode println.

Pour affiche « toto » sur écran on écrit **System.out.println("toto")** (La méthode *println* appartient à la classe System.out)

Exemple

```
public class Primitive
{
public static void main(String[] arg)
{
System.out.println("Primitives:");
int intA = 12;
System.out.println(intA);
int intB = intA;
System.out.println(intB);
intA = 48;
System.out.println(intA);
System.out.println(intB);

System.out.println("Objets");
Entier entierA = new Entier(12);
System.out.println(entierA);
Entier entierB = entierA;
System.out.println(entierB);
EntierA.valeur = 48;
System.out.println(entierA);
System.out.println(entierB);
}
}
class Entier
{
  int valeur;
   Entier(int v) {valeur=v;}
}
```

Affichage

Primitives 12 12 48 12
Objets 12 12 48 48

5 Notions de variables, classe et méthodes final

5.1 Variable final

Une variable déclarée *final* ne peut plus voir sa valeur modifiée (final int x=1 ;).

5.2 Méthodes final

Les méthodes *final* ne peuvent pas être redéfinies dans les classes dérivées.

5.3 Classes final

Une classe final ne peut être étendue pour créer des sous-classes.

6 Notions d'attributs et méthodes static

6.1 Attribut static

Un attribut statique (déclaré avec le mot clé **static** : *static int x=3*) est un attribut partagé par tout les objets de la classe (tout les objet ont la même valeur pour cet attribut).

6.2 Méthode static

Une méthode statique (déclarée avec le mot clé **static** : *public static void f( )*) est une méthode qui n'a accès qu'aux membres static de la classe.

7 Le mot clé this

this désigne une référence sur l'objet courant.

Exemple

```
Point(int _x, int _y)
{
        this._x = _x ;
        this._y = _y ;
}
```

Pour distinguer le paramètre du constructeur à l'attribut, on utilise le mot clé **this**

8 Mon premier programme en Java

Pour pouvoir exécuter un programme en Java, il faut définir une fonction particulière appelée : main. Cette méthode doit être contenue dans une classe (le tout objet), elle possède un en-tête obligatoire fixé :

public static void main(String[] arg)

static : signifie qu'elle ne pourra faire appel qu'aux champs statiques de la classe
String[] arg : tableau d'arguments du programme

Exemple

```
class Exemple
{
        public static void main(String[] arg)
        {
                System.out.println("Bonjour" ) ;
        }
}
```

9 Notion de Paquetage

Un paquetage ou package est un regroupement de classes. On regroupe dans un même paquetage des classes ayant une thématique et des fonctionnalités communes. Un paquetage contient des classes et des sous-paquetages.

Les classes de l'API appartiennent à plusieurs packages (Exemple : java.util, java.awt…). Pour utiliser une classe qui appartient à un package de l'API, on utilise l'instruction **import** suivi du nom du package.

Exemple

- Pour utiliser la classe Vector du package java.util on écrit : **import java.util.Vector ;** au tout début du programme.
- Pour utiliser toutes les classes du package java.util on écrit : **import java.util.* ;**

9.1 Création de paquetage

Si vous souhaitez qu'une classe que vous avez crée appartienne à un package particulier, vous devez le spécifier explicitement au moyen de l'instruction **package.** Il faudra aussi créer un répertoire au nom du package et sauvegarder les classes appartenant au package dans ce répertoire.

Exemple

```
package MonPackage

public class Employe
{
        …
}
```

La classe *Employe* doit être sauvegarder dans un répertoire appelé MonPackage.

9.2 Package et accessibilité

9.2.1 Portée d'une classe

La portée d'une classe est définie comme la zone dans laquelle elle est visible. Seules les classes publiques sont accessibles depuis l'extérieur du package.

Modificateur de portée	*Portée*
aucun	Le paquetage
public	Partout

9.2.2 Portée des membres d'une classe

***protected :** Les membres d'une classe peuvent être déclarés protected. Dans ce cas, l'accès en est réservé aux méthodes des classes dérivées, des classes appartenant au même package ainsi qu'aux classes appartenant au même package que les classes dérivées.
***private :** Accès au niveau de la classe
***public :** accès partout

Remarque

Un attribut sans modificateur (modificateur par défaut) n’est pas accessible depuis l’extérieur du paquet de la classe.

10 Les tableaux

10.1 Déclaration

Exemple:

int[] x où int x[];

x est un handle (référence) correspondant à un tableau qui contient des entiers.

10.2 Initialisation

```
int[] x;
x = new int[dimension];
```

x est un référence sur un tableau de taille égale à *dimension*

Remarques

* Les tableaux de littéraux peuvent contenir des variables.

Exemple:
```
int a =1, b= 2;
int [ ] y = {a,b};
```

- Si on ne fait pas d'initialisation, une initialisation automatique est alors faite (pour les tableaux de primitives à 0, pour les tableaux de handles à null)

10.3 Taille des tableaux

La taille d'un tableau est fixe. Néanmoins, il n'est pas nécessaire qu'elle soit connue au moment de la compilation.

Exemple

```
Import java.util.*;

class Test2
{
public static void main(String[] arg)

{

int x = Math.abs((new Random())%10;
int[] y;
y = new int[x] ;
System.out.println(x);
}
}
```

Dans ce cas, on peut connaître la taille d'un tableau avec le champs *length* (System.out.println(y.length)).

10.4 Les tableaux multi-dimensionnels

Syntaxe

```
int[][] x;
x = new int[2][4];
```

Initialisation

```
int[][] x = {{1,2,3,4}, {5,6,7,8}};
```

11 Les chaînes de caractères

En Java, les chaînes de caractères sont des objets. Ce sont des instances de la classe *String*. Java utilise une approche particulière, en effet, les contenus des chaînes de caractères ne peut plus être modifié une fois initialisé (Les chaînes de caractères se comportent comme des primitives).

Exemple

```
public class chaines
{
public static void main(String[] arg)

{
String chaineA = new String ("chaine1");
System.out.println(chaineA);

String chaineB = chaineA;
System.out.println(chaineB);

ChaineA = "chaine2";
System.out.println(chaineA);
System.out.println(chaineB);}}
```

Résultat

chaîne1 chaîne1 chaine2 chaine1

Bien que les chaînes sont des objets, ils se comportent ici comme des primitives. Cela est dû au fait que les chaînes ne peuvent être modifiées.

Remarque

Java dispose d'une autre classe, appelée StringBuffer, qui permet de gérer des chaînes dynamiques.

11.1 La classe "StringBuffer"

La classe StringBuffer permet de modifier des chaînes de caractères notamment grâce à la méthode append( ).

Exemple

```
StringBuffer chaine;
chaine = new StringBuffer ("Bon");
chaine.append("jour");
System.out.println(chaine);
```

Affichage

Bonjour

11.2 Quelques méthodes relatives à la classe "String"

Concaténation

Les chaînes de caractères peuvent être concaténées

Exemple

```
String chaine1 =  "Bonjour";
String chaine2 = "tout le monde";
chaine 1 = chaine1 + chaine2; // création d'une chaîne constante "bonjour tout le monde et affectation à la variable chaine1
```

La méthode length( )

length( ) retourne le nombre de caractères de la chaine.

La méthode equals( )

equals( ) permet de tester l'égalité de deux chaînes de caratères

Remarque

L'emploi de l'opérateur = = pour tester l'égalité de 2 chaînes n'est pas correct (= = teste l'égalité des références des variables et non celles de leurs contenus).

La méthode substring( )

La méthode substring ( ) extrait une sous-chaîne d'une chaîne donnée. La sous-chaîne commence à l'indice donné en premier argument et se termine à l'indice donné en second argument.

Exemple

```
String chaine;
String chaine1;
chaine1.substring(2,6);
```

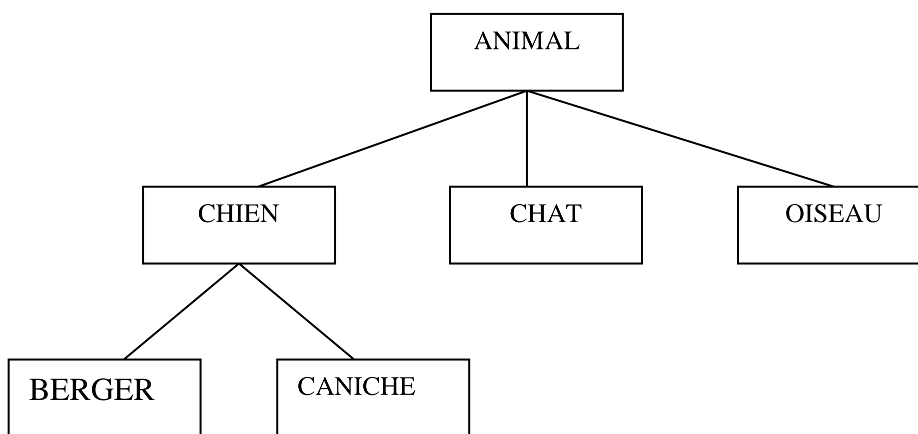
CHAPITRE 3 : L'HERITAGE

1 Propriétés de l'héritage en Java

- Supposons qu'on dispose de 2 classe A et B, si B hérite de A, on appelle B la sous classe et A la super classe.
- Une sous-classe possède tous les champs et toutes les méthodes de sa super classe à l'exclusion de ceux qui sont privés.
- Chaque classe ne peut avoir qu'une super classe (pas d'héritage multiple)

Pour définir une sous-classe, on utilise le mot clé *extends*

Exemple




```
class Animal

{
  private bool vivant ;
  private int age ;
  public Animal(int a)
  {
    age = a ;
    vivant = true ;
  }

public void veillir( )
{++ age ;
}
public void mourir( )
{ vivant = false ;
}

public void crier( )
{ }
}

Canari extends Animal
```

```
{
void crier()
 {
  System.out.println(‘’Coucouc’’) ;
}
```

La méthode crier( ) est redéfinie dans la classe Canari. La méthode crier( ) de la classe Animal est toujours accessible mais en la préfixant du mot clé ***super***.

Exemple

```
Class Canari extends Animal

{
 public Canari (int a)
 { super (a) ; }
}
```

2 Constructeur de sous-classe

Pour utiliser un constructeur dans une sous-classe, il faut nécessairement le redéfinir dans cette sous-classe.

Remarque

Si on n’invoque pas le constructeur de la classe mère, il faut nécessairement avoir un constructeur vide au niveau de la classe mère sinon le compilateur fera appel au constructeur vide par défaut. Si un constructeur est définis au niveau de la classe mère, le constructeur vide par défaut n’existe plus, si un constructeur vide n’est pas défini, il y aura une erreur de compilation.

Exemple

```
class Numero

{
 private int num ;

 public void plusUn( )
{ num ++ ;}
}

Numero n1 = new Numero( ) ;
n1.plusUn( ) ;
```

Maintenant si on définit un constructeur :

```
class Numero
{
private int num ;

public void plusUn( )
```

```
{ num ++ ;}

public Numero(int valeurDebut)
{ num = valeurDebut ;}
```

Si on écrit Numero n = new Numero( ) on aura une erreur ! !

Remarque

En java, il n'existe plus de notion d'héritage publique, privé ou protégé.

3 Surcharge d'une méthode héritée

La classe Canari redéfinit la méthode crier( ), on dit que la méthode crier est surchargée.

Exemple

```
Animal a = new Animal(5) ;
Canari c = new Canari(10) ;

a.crier( ) (on aura aucun affichage)
c.crier( ) (on aura un affichage)
```

On dit que la méthode est *polymorphe*. Elle a plusieurs formes, et à chaque appel, la forme à exécuter est choisie en fonction de l'objet associé à l'appel.

Remarque

Plus besoin de fonctions « virtual » comme en C++ pour que le polymorphisme se fasse. Java utilise en effet une liaison tardive (late binding)

3.1 Le « late binding »

Java utilise une technique particulière pour déterminer quelle méthode doit être appelée. Dans la plupart des langages, le lien entre l'appel et la méthode est établi au moment de la compilation. Cette technique est appelée « early binding », que l'on pourrait traduire par liaison précoce. Java utilise cette technique pour les appels de méthodes *final*. En revanche, pour les méthodes qui ne sont pas *final*, Java utilise la technique du late binding (liaison tardive). Dans ce cas, le compilateur n'établit le lien entre l'appel et la méthode qu'au moment de l'exécution du programme. Ce lien est établi avec la version la plus spécifique de la méthode.

4 Méthodes et classes abstraites

4.1 Méthodes abstraites

Une méthode est abstraite si seul son en-tête est donné, le corps est remplacé par un point virgule, la méthode est alors de type *astract* .

Exemple

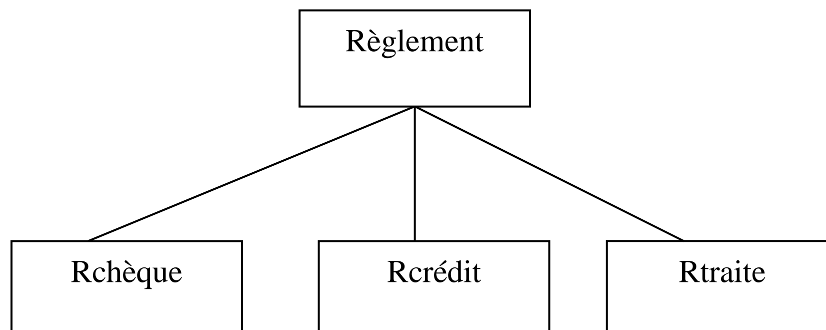
public abstract void resolution( ) ;

4.2 Classes abstraites

Une classe qui possède au moins une méthode abstraite est abstraite et doit être déclarée avec le modificateur : *abstract*.

Exemple

Intéressons-nous à la définition d'une classe dont les instances représenteront les règlements des factures envoyées aux clients d'une entreprise. Un client peut régler par chèque, par virement ou encore par traite.




```
abstract class Reglement
{
  abstract float credite( ) ;
}

 class Rchéque extends Reglement

{
.....

float credite( )
{......
ensembles d'instructions.......
 }

}
```

Une méthode abstract correspond à une propriété dont on veut forcer l'implémentation dans chaque sous-classe. Si une sous-classe ne redéfinit pas une méthode abstract, et si cette méthode n'est pas aussi abstract dans cette sous-classe on aura automatiquement une erreur.

5 Les interfaces

Java n'admet pas d'héritage multiples (héritage à partir de plusieurs classes). Pour répondre à certaines situations, Java permet une forme d'héritage multiple avec des classe très particulières que l'on appelle *interface*.

Les interfaces possèdent un niveau d'abstraction supérieur à celui des classes abstraites. Contrairement aux classes abstraites, les interfaces :

- Ne possèdent pas de champs, à l'exception des constantes (champs de classe constants),
- Ont des méthodes qui sont toutes implicitement abstraites (elles n'ont pas de corps),
- Comme pour les classes abstraites une interface ne peut être instanciée.

Une interface est déclarée grâce au mot clé *inteface*

Exemple

```
public interface UneInterface
{
 public void method1( ) ;
public double method2( ) ;
}
```

Une classe peut 'hériter' des caractéristiques d'une interface, on dira dans ce cas que la classe implémente l'interface car elle devra obligatoirement redéfinir toutes les méthodes de l'interface. Pour spécifier le fait qu'une classe implémente une interface, on utilise le mot clé *implements.*

Exemple

```
Public interface UneInterface
{
public void method1( ) ;
public void method2( ) ;
}

class UneClasse implements uneInterface

{
// champs.....
public void method1( )
{
// ensemble d'instructions
}
public void method2( )
{

//ensemble d'instructions

}

}
```

Remarque

- Une interface peut étendre une autre interface
- Une classe peut implémenter plusieurs interfaces, mais ne peut étendre qu'une seule classe (abstraite ou non).

CHAPITRE 4 : GESTION DES EXCEPTIONS

1 Comment gérer les erreurs d'exécution ?

Exemple

```
public class Ville
{
protected int nbHabitants ;

public void setNbHabitants (int unEntier)
 {
  if (unEntier <= 0) erreur(''un nbr d'habitants doit être >0'') ;
  nbHabitants = unEntier ;
  }

protected void erreur (String message)
  {
   System.out.println(''classe''+ getClass( ).getName( ) + '' :''+message) ;
  System.exit(1) ;
 }
}
```

La fonction getClass( ) renvoie un objet de la classe. La fonction getName ( ) renvoie le nom de la classe.

Sur l'exemple on peut considérer que l'arrêt est brutal. Pour cela on utilise la notion d'exception qui permet de gérer les erreurs.

2 Mécanisme des exceptions

une exception est un objet qui est instancié lors d'un incident : on dit qu'une exception est levée. Lorsqu'une exception est levée, le traitement du code de la méthode est interrompu et l'exception est propagée à travers la pile d'exécution de méthode appelée en méthode appelante. Si aucune méthode ne capture l'exception, celle-ci remonte jusqu'à la méthode du fond de la pile d'exécution : l'exécution se termine avec une indication d'erreur.

Déclaration

Exemple

```
class NbHabException extends Exception
{
 // Instructions…..
}

class Ville
{

 public void setNbHabitants (int unEntier) throws NbHabException
```

// throws signifie que la méthode est susceptible de lever une exception

```
{
if (unEntier<=0) throw nbHabException(unEntier); //lever une exception
```

3 Propager ou capturer une exception

Supposons que dans la classe Ville, on a un constructeur avec 2 arguments qui appelle la méthode SetNbHabitants( );

Exemple

```
public Ville(String nomVille, int population)
{
  this (nomVille);
 setNbHabitants(population);
}
```

On aura une erreur de compilation, en effet il faudra soit propager l'exception, soit la capturer.

Propager

On utilise le *throws*

```
public Ville(String nomVille, int population) throws NbHabException
{….
   }
```

Capturer

La capture d'une exception est effectuée avec les clauses *try* et *catch*

- La clause try définit un bloc d'instructions pour lequel on désire capturer les exceptions éventuellement levées.
- La clause catch définit l'exception à capturer, en référençant l'objet de cette exception par un bloc de paramètres puis le bloc à exécuter en cas de capture.

Exemple

```
void method( )
{
String nomV;
int nbh;
Ville v1;
try { v1 = new Ville(nomV, nbh); } //nous voulons capturer certaines exceptions, si elles sont levées où propagées lors de l'exécution du bloc qui suit try

catch(nbHabException nEx)

{
  System.out.println("Fichier Incohérent");
  System.out.println(nEx);
  System.exit(1);
}

//traitement de l'exception
```

CHAPITRE 5 : LES ENTREES / SORTIES

1 Généralités sur les flots de données

Les flots de données permettent la gestion des opérations de lecture et d'écriture sur ou vers des sources différentes.

Il existe quatre classes principales permettant de traiter les flots:

Flots d'octets

- InputStream
- OutputStream

Flots de caractères

- Reader
- Writer

Les noms des classes de flux (Héritant des classes abstraites InputStream, OutputStream, Reader et Writer) se comprennent assez aisément si l'on utilise ces termes comme suffixe permettant de déterminer la nature du flux. Pour la source ou la destination, on utilise les préfixes suivants:

Source ou Destination	Préfixe
Chaîne de caractères	String
Fichier	File
Flux d'octets	InputStream ou OutputStream

Exemple

StringWriter (Héritant de la classe Writer): flux de caractères en écritures, crée à paartir d'une chaîne de caractères.

FileInputStream (Héritant de la classe InputStream): flux d'octets en lecture, crée à partir d'un fichier.

InputStreamReader (Héritant de la classe Reader): flux de caractères en lecture construits à partir d'un flux d'octets.

2 Saisir des données envoyées par le clavier

Pour saisir les données envoyées par le clavier on utilise généralement deux classes appartenant au paquetage java.io: InputStreamReader et BufferedReader (flux de caractères en lecture construit à partir d'un Buffer).

Remarque

On utilise la classe *BufferedReader* car elle contient une méthode *readLine( )* permettant la lecture d'une chaîne de caractère. Avec la classe *InputStreamReader* on ne peut lire des caractères qu'un par un.

2.1 Construction d'objets *BufferedReader* et *InputStreamReader*

BufferedReader a un constructeur qui prend en argument une instance de *Reader* (dont hérite *InputStreamReader)*. *InputStreamReader* admet quant à elle un constructeur ayant comme argument un flot d'entrée (le flot d'entrée sur clavier est désigné par System.in).

2.2 Exemple de lecture à partir du clavier

```
String ligne;
BufferedReader fluxEntree = new BufferedReader( new InputStreamReader(System.in));
ligne = fluxEntree.readLine( );
```

Remarques!!

- La méthode *readLine( )* lance une exception du type IOException(Input Output Exception) qu'il faudra gérer.
- Pour pouvoir lire des types *doubles, int,…*il faudra utiliser la méthode readLine( ) pour lire des caractères et faire un cast avec les méthodes *valueOf( ) et doubleValue( )…*

Exemple complet

```
import java.io.*;
class Saisie
{
 public static void main(String[] arg)
  {
   String ligne;
   double x;
   BufferReader entree =  new BufferedReader (new InputStreamReader(System.in));

 try{
       ligne = entree.readLine( );
       x = Double.valueOf(entree).doubleValue( );
       System.out.println(x);
      }

catch(Exception e){
      System.out.println("Erreur de lecture");
      System.exit(1);
                           }
   }
}
```

3 Lire ou écrire des caractères dans un fichier

La classe *FileReader* permet de lire des caractères dans un fichier. Cette classe contient un constructeur qui a comme argument une chaîne de caractères représentant le nom du fichier.

La classe *FileWriter* permet d'écrire des caractères dans un fichier. Cette classe contient un constructeur qui a comme argument une chaîne de caractères représentant le nom du fichier.

Exemple

Supposons que l'on veuille écrire un programme permettant d'écrire une chaîne de caractères dans un fichier nommé *copie-essai.txt* puis copier le fichier *essai.txt* caractère par caractère dans notre fichier *copie-essai.txt*.

Nous allons utiliser pour cela les deux classes *FileReader* et *FileWriter*

```
import java.io.*;

class LireEcrireTexte
{
  public static void main(String arg[]) throws IOException
  {
    FileReader lecteur;
    FileWriter ecrivain;
    int c;

    lecteur = new FileReader("essai.txt");
    ecrivain = new FileWriter("copie-essai.txt");
    ecrivain.write("on essaye de faire une copie");

    while( (c = lecteur.read( ) ! = -1 )   ecrivain.write(c);
    lecteur.close( );
    ecrivain.close( );
   }
}
```

Remarque

La méthode read( ) renvoie –1 si on est en fin de flux

4 Ecrire et lire dans un fichier texte

4.1 Ecrire dans un fichier texte

Pour écrire dans un fichier texte (on écrit des chaînes caractères et plus des caractères seulement), on utilise généralement la classe *PrintWriter* pour pouvoir utiliser la méthode *println(String)* de la même façon qu'avec *System.out*. La classe *PrintWriter* possède un constructeur qui a comme argument un objet *Writer*.

Exemple

```
import java.io.*;
class EcrireFichierTexte
{
```

```
 public static void main(String arg[]) throws IOException
 {
   int n = 5;

   PrintWriter ecrivain = new PrintWriter(new  BufferedWriter( new FileWriter(arg[0])));
   ecrivain.println("Bonjour");
   ecrivain.println(n);
   ecrivain.println(new Integer(36));
   ecrivain.close( );
 }
}
```

Remarque

On aurait pu plus simplement initialiser *ecrivain* par: *ecrivain = new PrintWriter(new FileWriter(arg[0]));* mais alors les écritures n'utiliseraient pas de mémoire-tampon.

4.2 Lecture dans un fichier texte

Pour pouvoir lire des chaînes de caractères à partir d'un fichier on utilise généralement la classe *BufferedReader*. La classe dispose d'un constructeur qui admet comme argument un objet de type *Reader* (On utilise généralement un objet *FileReader*).

Exemple

```
import java.io.*;
class lireLigne
{
  public static void main(String[] arg) throws IOException
  {
   BufferedReader lecteurAvecBuffer = null;
   String ligne;

   try
   {
    lecteurAvecBuffer = new BufferedReader(new FileReader(arg[0]));
   }

  catch (FileNotFoundException e)
   {
    System.out.println("Erreur d'ouverture");
    }


   while ( (ligne = lecteurAvecBuffer.readLine( ) ) ! = null) System.out.println(ligne);
   lecteurAvecBuffer.close( );
  }
}
```

CHAPITRE VI: INTERFACES GRAPHIQUES

1 Introduction

Généralement en développement objet, on distingue l'interface graphique de la partie traitement. Cette séparation est préférable, elle respecte les principes de la programmation objet et simplifie grandement le travail de migration en cas de changement de l'interface graphique.

L'avantage apporté par Java en matière d'interface graphique est que toute interface développée sur une machine donnée avec un système d'exploitation donné, peut être portée sur une machine différente avec un autre système d'exploitation, et cela sans réécriture du code. L'aspect visuel de cette interface est très peu modifié par ce portage.

2 Applets et applications graphiques

Les programmes graphiques en Java se divisent en deux catégories: les applets et les applications graphiques.

2.1 Applet

Un applet est un programme qui s'exécute dans un navigateur Web, à partir d'un document HTML.

Avantage: Interactivité
Inconvénient: Sécurité

2.2 Applications graphiques

a Fenêtre vide

Les composants des applications graphiques sont rassemblés dans une bibliothèque nommée AWT.

Exemple

```
import java.awt.*;

public class Bonjour
{
 public static void main(String[] arg)
 {
   Frame fen;
   fen = new Frame("bonjour");
   fen.setSize(250,100);
   fen.setVisible(true);
 }
}
```

Remarque

- Pour le moment le seul moyen de fermer la fenêtre est de fermer la machine virtuelle CTRL+C
- On verra plus tard que les applets contrairement aux interfaces graphiques utilisent des Panel et non des Frame.

b Fenêtres contenants des messages

Exemple

```
import java.awt.*;
class SortieTexteGraph extends TextArea
{
  SortieTexteGraph(String texte, int nbLignes, int nbColonnnes, String police, int typePolice,
int taillePolice)
   {
    super(texte, nbLignes, nbColonnes, SCROLLBARS_NONE);
    setFont(new Font(police, typePolice, taillePolice));
    setEditable(false);
   }
 }
```

La classe SortieTexteGraph étend la classe TextArea qui est un modèle de composant graphique utilisé pour écrire du texte portant sur plusieurs lignes.

Le constructeur de TextArea est invoqué par super( ), en plus du message à afficher, il fixe le nombre de lignes et de colonnes que peut afficher ce composant. Le dernier argument du constructeur règle la présence ou l'absence des barres de défilement, ici nous avons choisi de ne pas faire apparaître (SCROLLBARS_NONE).

La police de caractères est crée grâce à setFont qui elle même utilise la méthode Font.

Enfin le message ne pourra pas être modifié par l'utilisateur dans la fenêtre graphique setEditable(false).

Exemple de classe utilisant SortieTextGraph

```
import java.awt.*;
public class MessageComposant1 extends Frame
{
  SortieTexteGraph zoneTexte;
  int nbLignes = 5;
  int nbColonnes = 20;

  public MessageComposant1 (String titre, String message, int taillePolice)
  {
   super(titre);
zoneTexte = new SortieGraph(message, nbLignes, nbColonnes, "Serif", Font.BOLD,
taillePolice);
```

```
add(zoneTexte, "North");
}

 public static void main(String[] arg)
 {
    MessageComposant1 fen;
    String titre = "Affichage dans une fenêtre texte";
    String mess = "Un composant \n TextArea \n permet l'affichage \n" + "de plusieurs lignes
\n";
    fen = new MessageComposant(titre, mess, taillePolice);
    fen.pack( );
    fen.setVisible(true);
   }
}
```

La méthode pack( ) détermine la taille de la fenêtre en fonction des éléments qu'elle contient.

Gestionnaire de mise en page

L'ajout d'une zone de texte à la fenêtre se fait sous le contrôle d'un gestionnaire de mise en page. Le rôle d'un gestionnaire de mise en page (Layout Manager) est de placer les composants graphiques dans la fenêtre.

Il existe cinq gestionnaires de mise en page, qui ont chacun des règles de mise en page qui leur sont propres:

- BorderLayout
- FlowLayout
- CardLayout
- GridLayout
- GridBagLayout

BorderLayout

Le gestionnaire attaché par défaut à une fenêtre de type Frame est un BorderLayout, dont la caractéristique principale est de distinguer cinq zones dans la fenêtre: nord, sud, centre, ouest et est.

Remarque

Nous pouvons insérer un nouveau gestionnaire en utilisant la méthode setLayout(new FlowLayout( )).

c Lecture et écriture de texte

La classe TextArea fournit un modèle de composant destiné à l'affichage ou l'édition de plusieurs lignes de texte.

La classe TextField quant à elle est dédiée à l'édition ou l'affichage d'une seule ligne de texte. Elle est bien adoptée à la saisie de valeurs numériques ou de textes brefs (entrées au clavier).

Exemple

```
public class TextFieldDemo extends Frame
{
  public TextFieldDemo( )
  {
    String titre = "Demonstration de textField";
    setTitle(titre);
    setLayout(new FlowLayout( ) );
    TextField zoneEntree = new TextField("Entrer un nombre dans ce champ");
    add(zoneEntree);
}
}
```

d Etiquettes (label)

Les composants de type label sont des étiquettes que l'on place à côté des autres composants (zone de texte, liste…) pour les légender.

Exemple

```
public class labelDemo extends Frame
{
  public labelDemo( )
  {
   String titre = "démonstration de label et de TextField";
   setTitle(titre);
   setLayout( new FlowLayout(  ) );
   label etiquetteZoneEntree = new label("coefficient du terme en x2");
   add(etiquetteZoneEntree);
   TextField zoneEntree = new TextField("3.14");
   Add(zoneEntree);
  }
}
```

e Buttons (Bouton)

Les composants de type Button sont des boutons classiques dont on se sert pour provoquer des actions.

Exemple

```
public class ButtonDemo extends Frame
{
  public ButtonDemo( )
  {
    String titre = "Démonstration de boutons"
```

```
    setTitle(titre);
    setLayout(new FlowLayout( ) );
    Button boutonValider = new Button("valider");
    add(boutonValider);
    Button boutonAnnuler = new Button("Annuler");
    Add(boutonAnnuler);
  }
}
```

f Menus

Les menus utilisent les classes Menu, MenuItem et MenuBar. Un objet de type Menu est un menu déroulant qui est attaché à une barre de menu (MenuBar) et comporte des éléments (MenuItem). MenuItem et MenuBar sont des sous-classes de MenuComposant et Menu une sous classe de MenuItem.

Exemple

```
public class MenuDemo extends Frame
{
  public MenuDemo( )
  {
    string titre = "demonstration menu";
    setTitle(titre);
    MenuBar barreMenu = new MenuBar( );
    setMenuBar( barreMenu); //ajoute la barre de menu à la fenêtre
    Menu menuFichier = new Menu("Fichier");
    Menu menuEdition = new Menu("Edition");
    Menu menuAide = new Menu("Aide");
    barreMenu.add(menuFichier);
    barreMenu.add(menuEdition);
    barreMenu.add(menuAide);
    MenuItem menuNouveau = new MenuItem("nouveau");
    MenuItem menuOuvrir = new MenuItem("ouvrir");
    menuFichier.add(menuNouveau);
    menuFichier.add(menuOuvrir);
  }
}
```

CHAPITR VII Les accès réseau en Java

1. Accès via une URL

Une URL contrairement au chemin d'accès classique à un fichier permet de désigner un fichier de manière uniforme quelque soit le système qui héberge ce fichier.

Une URL est composée :

- D'une chaîne représentant le protocole à utiliser au fichier (Exemple : http, ftp…) suivi du symbole :
- Le nom de l'hôte qui fournit le service recherché (Exemple : www.yahoo.fr)
- Le chemin d'accès au fichier recherché sur l'hôte. Les répertoires éventuels de ce chemin sont séparés par le symbole /

Remarque

Une URL peut représenter un fichier mais aussi de manière plus générale une ressource (par exemple un programme renvoyant un résultat)

2. Manipulation d'une URL

Pour manipuler une URL on va utiliser un certain nombre de classes appartenant au package *java.net*.

La classe URL

La classe URL permet de créer un objet permettant la manipulation d'une URL. Différents constructeurs sont définis dans la classe URL, nous pouvons par exemple utiliser le constructeur suivant :

public URL(String s) throws MalformedURLException

Ce constructeur contient comme paramètre une chaîne de caractères qui représente le chemin d'accès au fichier. Ce chemin contient le protocole d'accès, l'adresse de l'URL et le chemin d'accès au fichier (exemple : http://www.isaip.uco.fr/MonRepertoire/MonFichier). Les différents constructeurs d'URL sont susceptibles de lever une exception du type *MalformedURLException* si l'accès à l'URL n'est pas possible.

Plusieurs méthodes sont définis dans la classe URL, ces méthodes permettent de lire des données, de fournir le protocole d'accès, de fournir le nom de l'hôte…

public String getProtocol() → renvoie le protocole d'accès
public String getHost() → renvoie le nom du hôte
public final inputStream openStream() throws IOException → permet de lire des données à partir de l'URL

Exemple

Le programme suivant permet de lire le contenu d'une URL et de l'afficher sur la sortie standard.

```
import java.io.*;
import java.net.*;

class LectureURLSansConnexion
{
	public static void main(String[] arg)
	{
		try
		{
			URL y = new URL("http://www.isaip.uco.fr/");
			DataInputStream dis = new DataInputStream(y.openStream());
			String inputLine;
			while( (inputLine=dis.readLine()) != null)
							System.out.println(inputLine);

			dis.close();
		}

		catch(MalformedURLException me)
		{
			System.out.println("Erreur de connection à l'URL");
		}

		catch(IOException ioe)
		{
			System.out.println("Erreur d'ouverture du flux");
		}
	}
}
```

3. Connexion à une URL

Nous pouvons ouvrir une connexion avec une URL en utilisant la méthode *openConnection()* de la classe URL.

public URLConnection openConnection() throws IOException

La méthode openConnection() renvoie un objet de la classe *URLConnection*. Cette classe est abstraite et permet de gérer la connexion avec l'URL. Les méthodes de la classe URLConnection permettent de lire, écrire dans le fichier désigné par l'URL, mais aussi d'obtenir plein d'informations sur ce fichier (taille, date de création,…).

Pour ouvrir une connexion avec l'URL nous utilisons la méthode *connect()* d'URLConnection.

<u>Lecture et Ecriture dans une URL en utilisant URLConnection</u>

Nous avons vu qu'en utilisant la classe URL nous pouvons lire le contenu d'une URL. Avec URLConnection nous pouvons non seulement lire le contenu d'une URL mais aussi l'interroger sur son contenu et y écrire.
Pour lire et écrire avec *URLConnection* nous utilisons les fonctions suivantes :

public InputStream getInputStream() throws IOException
public OutputStream getOutputStream() throws IOExceptio

Exemple

Le programme suivant permet de lire le contenu d'une URL et d'afficher son contenu sur la sortie standard en utilisant la classe URLConnection.

```
import java.net.*;
import java.io.*;

class LectureURLAvecConnexion
{
	public static void main(String[] arg)
	{
		try
		{
			URL y = new URL("http://www.yahoo.fr/");
			URLConnection c = y.openConnection();
			DataInputStream dis = new DataInputStream(c.getInputStream());
			String inputLine;
			while ((inputLine = dis.readLine())!= null)
			{
				System.out.println(inputLine);
			}
			dis.close();
		}
		catch(MalformedURLException me)
		{
			System.out.println("Pb URL!!!");
		}

		catch(IOException ioe)
		{
			System.out.println("Erreur lecture");
		}
	}
}
```

CHAPITRE VIII : LES THREADS

Qu'est ce qu'un Thread ?

Un processus ou tâche est un programme en cours d'exécution. Les systèmes d'exploitation modernes sont presque tous multitâches. Ils sont conçus de façon que plusieurs programmes indépendants puissent s'exécuter en même temps. Cette simultanéité de l'exécution des programmes n'est généralement qu'apparente. Si l'on y regarde de plus prés, le système d'exploitation attribue, à tour de rôle, le processeur à chaque processus. Cette attribution est faite pendant de courts intervalles de temps. Elle donne l'illusion d'un déroulement continu et simultané des processus. Les processus sont indépendants les uns des autres, ils ont chacun leur propre espace mémoire. A l'inverse quand un programme crée des Threads (appelés processus légers), ils se partagent l'espace de travail du processus qui leur a donné le jour. On est naturellement amené à créer des Threads pour des travaux en interaction, même faible. Dans le cas contraire, il vaut mieux réaliser des programmes distincts.

Comment réaliser un Thread ?

La réalisation d'un Thread se fait très simplement en concevant une sous-classe de la classe *Thread* (appartient au package java.lang). Cette classe doit être dotée d'une méthode nommée *run()* qui est appelée lors du démarrage du Thread. Pour lancer un Thread, il faudra appelée la méthode *start()* à partir d'un objet issu de la sous-classe de Thread qu'on a crée.

Exemple

```
class MonThread extends Thread
{
	public MonThread()
	{
		}
	public void run( )
	{
		for(int i = 1 ; i<= 100 ; i++)
			System.out.println(i) ;
	}

	public static void main(String[] arg)
	{
		new MonThread().start() ;
		new MonThread().start() ;
	}
}
```

Pour renvoyer le nom du Thread nous pouvons utiliser la méthode *getName()* de la classe Thread.

Contrôler le déroulement d'un Thread

Plusieurs méthodes peuvent être invoquées pour contrôler le déroulement d'un Thread :

wait() : mise en attente du Thread
sleep(int) : interrompt le déroulement pendant une durée qui peut être spécifiée en paramètre

yield() : interrompt le déroulement afin de laisser du temps pour l'exécution des autres Threads

L'interface Runnable

En étendant la classe Thread on s'interdit de pouvoir étendre une autre classe puisque l'héritage multiple n'est pas autorisé en Java. Cette limitation est parfois fort gênante, c'est la raison pour laquelle il existe une autre façon de procéder. Pour qu'une classe n'héritant pas de la classe Thread ait les mêmes propriétés d'un Thread elle doit implanter l'interface *Runnable*. On la dote alors d'une méthode run(), ce qui est la seule exigence de cette interface.

Exemple

```
class MonThread2 implements Runnable
{
	public void run()
	{
		for(int i=1; i<=100; i++)
			System.out.println(i+" "+ Thread.currentThread().getName());

	}

	public static void main(String[] arg)
	{
		MonThread2 t1 = new MonThread2();
		MonThread2 t2 = new MonThread2();
		Thread unThread1 = new Thread(t1);
		Thread unThread2 = new Thread(t2);
		unThread1.start();
		unThread2.start();
	}
}
```

La synchronisation

Quand plusieurs Threads font appel, en même temps, à un seul objet il peut y avoir alors des comportements imprévisibles résultants de conflits d'accès à cette ressource. Cette situation est bien illustrée par l'exemple classique de retraits simultanés sur un compte bancaire. Plusieurs personnes retirent, en même temps, dans des agences différentes une somme correspondant au solde d'un même compte. Il y a alors, un risque que ce compte passe au rouge. Pour l'éviter, il faut que le compte soit verrouillé du début à la fin d'un retrait. Toute autre opération simultanée est alors interdite. Pour poser un verrou sur un objet, on déclare des méthodes synchronisée, on utilise pour cela le mot clé *synchronized*.

Exemple

```
public synchronized void retirerArgent(float somme)
```

Quand un Thread invoque une méthode synchronisée d'un objet, il pose un verrou sur cet objet qui ne sera libéré qu'à la fin de la méthode. Si un autre Thread invoque une méthode synchronisée d'un objet, il pose un verrou sur cet objet qui ne sera libéré qu'à la fin de la méthode. Si un autre Thread invoque une méthode synchronisée de cet objet, il sera bloqué jusqu'à la levée du verrou posé par le Thread précédent. Ce mécanisme empêche les accès concurrents à un objet.