Java concentré sucré

Petit opuscule de concentré Java destiné aux étudiants ayant des notions de programmation.

Daniel Tschirhart

Un objet est une capsule logicielle oblative avec un tropisme conatif dont l'hétéronomie est la marque de la durée de l'éphémère et de la hoirie.
Serge MIRANDA

Introduction à La Programmation Orientée Objet 1
Java et la POO 1
Les classes 1
Que contient une classe 1
Une classe contient des déclarations d'attributs 1
Une classe contient des définitions de méthodes. 1
Portée des attributs et des méthodes 2
Quelle portée utiliser ? 2
Attribut et méthodes d'instance, attribut et méthode de classe 2
Quand faut-il rendre un attribut ou une fonction statique 3
Règles de nommage des classes, attributs et méthodes (important) : 3
Portée d'une classe 3
Comment définir une classe 3
Exemple de classe 3
Syntaxe des accès aux attributs et aux méthodes. 4
Compiler une classe 4
Exécuter un programme Java 5
Exercices 5
Les données manipulées par Java 7
Types de données manipulées par Java 7
Données de type valeur 7
Données de type référence 7
Accès aux attributs et méthodes (rappel) 7
Au sein de la classe 7
En dehors de la classe 7
Création (instanciation) des objets 7
Représentation textuelle d'un objet 7
Destruction des objets 8
Point d'entrée d'un programme Java 8
Où commence un programme Java ? 8
Exercices 8
Ecriture et exécution des programmes Java 9
Edition des programmes Java (rappel) 9
Compilation des programmes Java (rappel) 9
Ligne de commande de compilation 9
Byte code Java 9
Chargement des objets en mémoire 9
Paramétrage des chemins de compilation du JDK et des exécutables du JRE 9
Rendre un programme java exécutable : archive JAR 10
Créer un JAR à l'aide d'eclipse 12
Documenter vos programmes avec la JavaDoc 13
Qu'appelle-t-on documenter les programmes ? 13
La JavaDoc 13

Que peut-on documenter 13
Générer la documentation 13
Exemple de documentation 13
Une classe très utilisée : La classe String 15
Présentation 15
Constructeurs principaux 15
Principales méthodes 16
Manipulation des caractères de la chaîne 16
Comparaisons et tests 17
Recherches 17
Conversions 18
La Classe StringBuffer 18
Présentation 18
Constructeurs principaux 18
Principales méthodes 18
transpypage des primitifs 19
Affectations entre primitifs de type différents 19
Conversion de primitifs en objet à l'aide de leur classe enveloppe 19
Nécessité des conversions 19
Exemple 19
Formatage des données 21
Formatage des flux 21
Formatage des chaines de caractères 21
Exemple 21
Opérations E/S console 23
Entrées de données à partir du clavier 23
Exemple d'utilisation 23
Mécanismes de réutilisation 25
Réutilisation ! 25
L'agrégation 25
La composition 26
L'héritage 28
Comment fait-on ! 28
Classes abstraites et methodes virtuelles pures 29
Classes interface 31
Gestion des erreurs d'exécution : exceptions 33
Gérer les erreurs d'exécution 33
Gestion des erreurs à base d'exceptions 33
Mise en place d'un mécanisme de gestion d'erreurs à base d'exceptions 33
Esquiver une ou plusieurs exceptions 34
Le langage des expressions régulières 35
Syntaxe des motifs d'expression régulière 35
Liste des méta caractères 35

Séquence d'échappement ___ 35
Répétitions ___ 35
Frontières de recherche ___ 36
Classes de caractères ___ 36
Règles de constructions des classes de caractères personnalisées ___ 36
Groupes ___ 36
Quantificateurs ___ 37
Mise en œuvre des expressions régulières dans la classe String ___ 37
Mise en œuvre des expressions régulières dans la classe `Pattern` ___ 37

Introductions aux Collections (conteneurs) ___ 41

Types de collection ___ 41

Collections séquentielles ___ 41
Vecteurs (Array) ___ 41
Pile ___ 41
File fifo ___ 41
Sacs (bags) ___ 42
Ensembles (set) ___ 42

Collections associatives ___ 42

Structures de données utilisées pour fabriquer les collections (structures de données fondamentales) ___ 42
Tableau ___ 42
Listes liées simples ___ 42
Listes doublement liées ___ 43
Arbres binaire ___ 44
Tables à adressage dispersé (hastable) ___ 44

Type de données abstrait et structures de données fondamentales ___ 45

Les classes paramétrées ___ 47

Nécessité de transtyper les types *Object* ___ 47

Utilisation des classes génériques ___ 47

Boxing et unboxing automatique ___ 48

Définition d'une classe générique ___ 48

Instanciation d'une classe générique ___ 48

Les Collections séquentielles dans Java ___ 49

Graphe d'héritage simplifié des collections séquentielle Java ___ 49

Parcourt des collections ___ 49

Interface List (T.D.A.) ___ 50

Classes concretes Vector, ArrayList, LinkedList (S.D.F.) ___ 50

Les ensembles : Interface Set (T.D.A.) ___ 51

Classe concrètes HashSet, TreeSet (S.D.F.) ___ 52

Les Collections Java associatives ___ 53

Définitions ___ 53

Hiérarchie des classes des conteneurs associatifs ___ 53

Interface MAP (T.D.A.) ___ 53

Classe TreeMap (S.D.F.) ___ 54

Classe HashMap (S.D.F.) ___ 54

Les opérations Entrées/sortie ___ 55

Les flots Java ___ 55
Construire les flots Java ___ 55
Utiliser les flots Java ___ 56

Flots prédéfinis 57
Les fichiers 57
Instancier des objets fichiers 57
Chemin des fichiers. 57
Fichiers d'octets 57
Fichiers de caractères 58
Fichiers de caractères lecture/ecriture lignes par lignes 58
Fichiers d'éléments primitifs 58
Fichiers d'objets 59
La programmation graphique 61
API graphiques de Java 61
Classes principales de l'API AWT 61
Création d'une fenêtre 61
Principaux éléments d'IHM 62
Les labels 62
Les zones d'édition 63
Les cases à cocher 64
Les boutons 65
Les combos 65
Les boites à listes 66
Les TextArea 67
FileDialog 68
Disposer les contrôles dans la fenêtre 70
Exemple 70
Mettre en place les contrôles dans la fenêtre 70
Exemple 70
Caractéristiques communes aux contrôles 70
Couleur d'avant plan et arrière plan 70
Taille du contrôle (Largeur, Hauteur) 70
Nom interne 70
Position et dimension 70
Police 71
Interagir avec l'utilisateur : la programmation événementielle 72
Principe de la programmation événementielle 72
Gestionnaires d'événements ou écouteur 72
Type d'événements 72
Abonnement aux événements 72
Syntaxe des abonnements 72
Quels événements pour quels contrôles ? 73
Evénements liés à la fenêtre 73
événements et abonnements 76
Contrôles de type Timer 78
Utilisation de Windows Builder sous Eclipse. 81
Création d'une Application avec WB 81
Ajout d'une IHM dans l'application précédente 82
Enrichissement de la fenêtre 83
Les applettes 85
Création 85
Sécurité des applettes 85
Les applications Java WebStart 87
Création d'une application Java WebStart 87
Java et La programmation Multitaches 89
Les tâches (thread) 89

Définition 89

Etats d'une tâche 89
Etat Non créé 89
Etat Dormant ou créé 89
Etat Prêt ou éligible 90
Etat en Exécution 90
Etat Bloqué ou Suspendu 90

Transitions entre états d'une tâche 90
Création d'une tâche (crée) 90
Activation d'une tâche (active) 90
Attribution du processeur à la tâche 90
Suspension d'une tâche 90
Reprise d'une tâche bloquée 90
Fin d'une tâche ou destruction d'une tâche 90
Suppression d'une tâche 90
Attribution du processeur physique 90
Abandon du processeur physique 90

Politiques d'ordonnancement 90

Les Taches dans Java 92
Package 92
Priorités des tâches 92
Création des tâches 92

Problèmes posé par les accès concurrents 94
Réentrance des fonctions 94
Sections critiques 94
Objets utilisés pour le verrouillage des section critiques 95

Modèles d'échange de donnée 96
Modèle producteur-consommateur 96
Modèle lecteur - rédacteur 100

Synchronisations des tâches 100
Mécanismes de synchronisation directs 100
Mécanismes de synchronisation indirects 100

La programmation réseau 103

Serveurs et clients 103

Exemple un intranet simple 103

Identification des clients : adresse MAC 103

Protocole IP : trames IP 103

Protocole TCP-IP 103

Adresses IP (V4) 103

Exemple de réseau 104

Classes de réseau 104

Masques de sous-réseau 104

Adresses réservées 105

Programmation réseau 105

Sockets et ports 105

Utilisation des sockets par le client 105
Connexion à un serveur 105
Exemple appel du service date avec Java 106
Les adresses Internet 106
Quelques exemples d'applications réseau 107

Utilisation des sockets par un serveur 111

Implémentation d'un serveur 111

Serveur multitâches (multithread) ______ **112**
Timeout des sockets ______ **113**

Page blanche

INTRODUCTION A LA PROGRAMMATION ORIENTEE OBJET

La **P**rogrammation **O**rientée **O**bjet vise à appliquer au domaine du logiciel les mêmes concepts de standardisation que ceux utilisés dans le domaine industriel (mécanique, électrique, électronique, …). Les objets sont des composants logiciels assemblés à partir d'autres composants en réutilisant directement les propriétés et des méthodes (agrégation ou composition) et ou spécialisés à partir d'autres composants (héritage des propriétés et des méthodes).
Les objets communiquent entre eux par envois de messages (appels de fonctions ou méthodes) et possèdent des propriétés reflétant leur état.
Les objets sont décrits à l'aide d'une classe. Attention une classe n'est pas un objet tout comme la recette du soufflé aux pommes n'est pas le soufflé aux pommes !

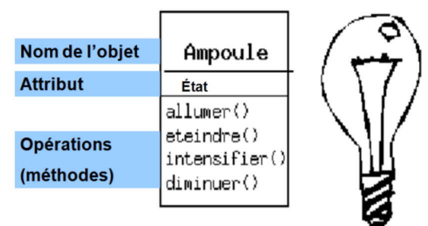
Il faut penser à un objet comme à une variable améliorée :

- Un objet stocke des données (attributs),
- Mais en plus on peut lui demander de faire des opérations sur lui-même.

En théorie, on peut prendre n'importe quel composant conceptuel d'un problème et le représenter en tant qu'objet dans un programme.

La classe ci-contre décrit une ampoule électrique :

- Son état (allumé, éteint) est mémorisé dans l'attribut *Etat*
- Les opérations que l'on peut faire avec cette ampoule (messages que l'on peut lui envoyer) sont définies dans la classe.



Une fois la classe *Ampoule* définie, on peut créer (on dit instancier) autant d'objets qu'on à l'aide de cette classe :

- `Ampoule lampe1 = new Ampoule()`
- `Ampoule lampe2 = new Ampoule()`

puis envoyer à chaque ampoule un message :

- `lampe1.allumer();`
- `lampe2.allumer();`

Lampe1

Lampe2

JAVA ET LA POO

Java reprend intégralement les concepts de la **P**rogrammation **O**rientée **O**bjet. On dit que Java est un langage pur objet car il n'autorise pas, contrairement au langage C++ dont il reprend la syntaxe, l'écriture de programmes non objet.
Les objets sont décrits à l'aide de classes (la recette) et sont créés (on dit instanciés) avant leur utilisation à l'aide de l'opérateur `new`.

LES CLASSES

Une classe est la description d'un objet. Un programme Java utilise pour l'essentiel des classes toutes faites issus de la bibliothèque Java. Ces classes sont rassemblées par catégorie (packages) et ces packages doivent être spécifiés avant leur utilisation par l'instruction `import` sauf pour le package `java.lang` qui est utilisé par défaut. `java.lang` contient les définitions des classes les plus usuelles.

QUE CONTIENT UNE CLASSE

UNE CLASSE CONTIENT DES DECLARATIONS D'ATTRIBUTS

Un attribut peut être :

- soit une variable (ou une constante) de type natif au langage (booléen, caractère, entier, réel). Ces variables sont dites de **type primitif**. Une variable n'est pas obligatoirement initialisée auquel cas elle prend la valeur *false* pour les booléen et 0 pour les autres types.
- soit une référence vers autre objet. Une référence n'est pas un objet mais une sorte de télécommande permettant de manipuler l'objet référencé. L'objet référencé est mémorisé dans une zone de mémoire spéciale appelée TAS gérée par la machine Java (le programmeur généralement ne manipule pas le TAS). La référence n'est pas initialisée (sauf initialisation explicite) et prend la valeur `null` (pour un attribut) et ne référence aucun objet.

UNE CLASSE CONTIENT DES DEFINITIONS DE METHODES.

- Une méthode est une fonction ou une procédure. Une fonction est un sous-programme retournant une valeur, une procédure n'en retourne pas. Fonctions et procédures peuvent avoir des paramètres composés de données de type primitif et/ou de références d'objets. Lors de la définition, ces paramètres sont appelés paramètres formels.
- Un ou plusieurs **constructeurs.**

Un constructeur est une méthode spéciale portant le nom de la classe et ayant pour but d'initialiser (on dit construire) l'objet. Le constructeur est appelé lors de l'instanciation de l'objet. Le constructeur utilisé est déterminé par le nombre et les types de ces paramètres précisé lors de l'instanciation de l'objet.
Si une classe ne définit aucun constructeur, un **constructeur par défaut** (constructeur sans paramètre) est automatiquement défini. Dans le cas contraire, le constructeur par défaut n'est pas défini (sauf définition explicite).

PORTEE DES ATTRIBUTS ET DES METHODES

La portée est spécifiée en préfixant les attributs et méthodes à l'aide des mots clés <rien>, `private`, `protected`, `public`.

- pas de préfixe : portée locale au package. Equivalent à `public` mais limité au package.
- préfixe `private` : portée locale à la classe.
- préfixe `protected` : portée locale à la classe et à la classe héritante (héritage voir plus loin).
- préfixe `public` : la portée est globale.

QUELLE PORTEE UTILISER ?

Un des principes de la programmation orientée objet consiste à masquer au maximum les détails d'implémentation, c'est-à-dire rendre l'utilisation d'une classe indépendante de son implémentation (codage). En pratique on doit pouvoir modifier une classe sans affecter les programmes utilisant cette classe. Cette dernière remarque s'applique parfaitement à la bibliothèque Java : un programme écrit pour une version 1.2 du JRE, s'exécute avec la bibliothèque 1.6 (le contraire n'est cependant pas vrai). Les règles à respecter sont les suivantes :

- Les attributs doivent êtres dans la majorité des cas privés (*`private`*). Pour accéder à un attribut depuis l'extérieur de la classe créer un assesseur, c'est-à-dire une fonction permettant de lire et/ou de modifier l'attribut (voir plus loin).
- Les méthodes accessibles depuis l'extérieur de la classe sont soit publiques, si l'accès doit se faire en-dehors de package, soit sans spécification de portée lorsque l'accès doit être locale au package (cas très particulier).
- Les méthodes exclusivement utilisées par les fonctions membre de la classe doivent avoir une portée `private`.
- Un constructeur doit avoir une portée `public`.
- Si une méthode ou un attribut doit être hérité (voir plus loin), la portée doit être spécifiée avec l'attribut `protected`.

ATTRIBUT ET METHODES D'INSTANCE, ATTRIBUT ET METHODE DE CLASSE

Par défaut les attributs et les méthodes sont liés à un objet et sont appelées attributs et méthodes d'instances. Ceci a pour conséquence que chaque objet issu de la même classe possède des attributs indépendants.
Il est possible casser l'affinité que possède un attribut ou une méthode avec un objet à l'aide du mot *static*. Cet attribut ou cette méthode est alors lié à la classe et non à l'objet dont il est issu et **devient commun** à tous les objets.
Les attributs et méthodes deviennent alors des attributs et des méthodes de classe.
L'utilisation du préfixe statique n'est pas anodin et a des conséquences importantes :

- Les fonctions de classes (fonctions statiques) ne peuvent plus accéder aux attributs et aux méthodes d'instances.
- La durée de vie des attributs devient celle du programme et non celle d'un objet particulier.
- Méthode ou attribut peuvent être accédés sans utiliser d'objet en utilisant l'opérateur *point*.

En résumé,

- les attributs statiques ont pour caractéristique principale suivante :
  - Ils existent en dehors de tout objet et sont commun à tous les objets.
  - Leur durée de vie est celle du programme.
- Les attributs d'instance ont pour caractéristique suivante :
  - Ils existent uniquement au sein d'un objet et leur valeur est mémorisée à l'intérieur d'un objet.
  - Leur durée de vie est celle de l'objet.
- Les fonctions ou méthodes statiques ont pour caractéristiques :
  - De ne pas nécessiter d'objet pour y accéder.
  - Ne peuvent accéder qu'à des attributs statiques.
- Les fonctions ou méthodes d'instance ont pour caractéristique :
  - Nécessitent d'instancier un objet pour pouvoir les utiliser.

QUAND FAUT-IL RENDRE UN ATTRIBUT OU UNE FONCTION STATIQUE

Jamais ou presque ! Il est très rare d'utiliser des attributs ou des fonctions statiques. On utilisera des fonctions statiques lorsque l'on veut regrouper au sein d'une classe un ensemble de fonctions utilitaires utilisées dans le même domaine. Un exemple typique est la classe `Math`. Cette classe regroupe l'ensemble des fonctions mathématiques plus courantes. Il parait logique que pour utiliser ces fonctions, il ne soit pas nécessaire d'instancier un objet. Exemple : calculer la racine carrée de 2 s'écrit :

```
double racineCarréDeDeux = Math.sqrt(2);
```

et non :

```
Math objetMath = new Math();
double racineCarréDeDeux = objetMath.sqrt(2);
```

Le même raisonnement est valable pour des attributs ou des constantes :

```
// Calcul du périmètre d'un cercle de 10m de rayon
double périmetreCercle = 2 * Math.PI * 10;
```

Les attributs statiques possèdent une propriété intéressante : ils sont commun à tous les objets et peuvent être utilisés pour partager un ensemble de valeurs communes.

REGLES DE NOMMAGE DES CLASSES, ATTRIBUTS ET METHODES (IMPORTANT) :

- Une classe <u>commence toujours par une majuscule</u>. Si la classe est un mot composé, la première lettre du mot composé est en majuscules. Exemple `Moteur, MoteurAvion`
- Un attribut <u>commence toujours par une minuscule</u>. Si l'attribut est un mot composé, la première lettre du mot composé est en majuscules. Exemple : `tableau`, `unTableau`. <u>Une constante est toujours en majuscules</u>.
- Une méthode <u>commence toujours par une minuscule</u>. Si la méthode est un mot composé, la première lettre du mot composé est en majuscules.

Bien que non obligatoire, le respect de ces règles :

- **Dénote la rigueur du programmeur**,
- Facilite la lecture des programmes, toutes les classes, méthodes et attributs de la bibliothèque Java respectant cette règle.

PORTEE D'UNE CLASSE

Si la classe est préfixée avec l'attribut `public` sa portée est globale, sinon elle est locale au package dans laquelle elle a été définie.

COMMENT DEFINIR UNE CLASSE

La classe minimale que l'on peut définir est la suivante (pas d'attribut, pas de méthode) :

```
public class Minimale
{

}
```

<u>Cette définition doit être placée dans un fichier portant le même nom que la classe et portant l'extension `.java`.</u>
<u>Un fichier `.java` peut comporter plusieurs classes (non conseillé), mais une seule doit posséder l'attribut `public`.</u>

Exemple : Dans l'exemple précédent la classe `Minimale` est placée dans le fichier `Minimale.java`

Cette classe possède une méthode implicite:

```
public Minimale() {}
```

Cette méthode est appelé **constructeur par défaut.** Celui-ci est générée automatiquement par le compilateur Java (sauf si la classe comporte déjà un constructeur). Un constructeur joue un rôle de première importance dans une classe puisqu'il a pour but de créer l'objet. La classe `Minimale` est parfaitement instanciable par :

```
Minimale mini = new Minimale();
```

Remarquer ici l'appel explicite du constructeur par défaut : `Minimale()`.
La classe ne présente évidement aucun intérêt.

EXEMPLE DE CLASSE

La classe suivante décrit un étudiant (en caractères **gras**, les mots réservés du langage).
<u>Cette définition doit être placée dans un fichier portant le même nom que la classe et portant l'extension `.java`</u> (rappel).

```
public class Etudiant
{
    private static int nombre;    // attribut de classe, portée globale
    private int promo;            // attribut d'instance, "    limitée à la classe
    private String nom ;          // attribut d'instance, "        "          "
            String école;         // attribut d'instance, portée limitée au package

    public static int getNombre () // méthode de classe de portée globale
    {
      return nombre;
    }
    public String toString()      // méthode d'instance de portée globale
    {
      return nom + ":" + promo;
    }
    int quellePromo()             // méthode d'instance, porté limité au package
    {
       return promo;
    }
    public Etudiant()               // constructeur par défaut
    {
       promo = 2017;
       nom = "";
    }
    public Etudiant(String n, int p, String école)// constructeur spécifique
    {
       nom = n; promo = p;
       this.école = école;
       nombre++;
    }
}
```

Remarque :

- L'attribut `école` a une portée incorrecte (choisir `private`).
- La méthode `getNombre` est un assesseur, il permet de lire l'attribut.

SYNTAXE DES ACCES AUX ATTRIBUTS ET AUX METHODES.

- Pour un élément de classe (attribut statique) : *`Classe.nom`*.
  Exemple : `System.out.println(`**`Etudiant.getNombre()`**`);`
- Pour un élément d'instance : *`objet.nom`*.
  Exemple :

```
Etudiant e1 = new Etudiant();
Etudiant e2 = new Etudiant(2014, "LaJoie", "Eigsi");
e1.école = "Eigsi";        // accès direct à un attribut : non recommandé
String s = e1.toString(); // appel d'une méthode
```

COMPILER UNE CLASSE

Une classe est compilée par le compilateur java `javac`. Celui-ci génère un fichier `.class` par classe compilée. Ce fichier est appelé *byte code* et peut être exécuté sur toute plateforme comportant le JRE (environnement java JRE : **J**ava **R**untime **E**nvironnement). Le JRE fournit un interpréteur appelé `java` qui exécutera vos fichiers `.class`.

Exemples (on supposera que les variables d'environnement sont correctement définies. Pour définir les variables d'environnement voir plus loin) :

- Lancer un shell (ouvrir une fenêtre de commande).
- Compiler la classe `Etudiant`
  `javac Etudiant.java`
  `javac` génère un fichier `Etudiant.class`
- Compiler toutes les classes (`A.java, B.java, C.java`) du répertoire courant
  `javac *.java`
  `javac` génère les fichiers `A.class, B.class, C.class`

EXECUTER UN PROGRAMME JAVA

Pour qu'un programme Java soit exécutable, une des classes doit comporter la fonction `main` suivante.

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

Pour exécuter un programme Java, il faut lancer l'interpréteur Java sur la classe contenant une fonction `main`. Exemple :

`java A` Exécute le *byte code* `A.class.` Celui-ci doit contenir la définition de la fonction `main.`

EXERCICES

1. Quelles sont les conventions de nommage des classes, des attributs et des méthodes Java ?
2. Quelle est la différence entre une classe et un objet ?
3. Qu'est-ce qu'un attribut de classe ?
4. Comment accéder à un attribut de classe `public` en dehors de sa classe ?
5. Comment accéder à un attribut de classe dans sa classe ?
6. Qu'est-ce qu'un attribut d'instance ?
7. Comment appeler une méthode `public` d'instance en dehors de sa classe ?
8. Comment appeler une méthode `public` de classe en dehors de sa classe ?
9. Quel type d'attribut peut utiliser une méthode de classe ?
10. Rappeler les règles d'utilisation des portées pour un attribut ou une méthode.
11. Le code suivant de la fonction `main` défini dans la classe `Etudiant` est faux !

```
1.  public static void main(String[] args) // défini dans la classe Etudiant
2.  {
3.    nom = "Durand";
4.    promo = 2012;
5.    Etudiant e1 = new Etudiant("Insa", 2012);
6.    System.out.println(e1.nom);
7.  }
```

- Donner les N° de lignes comportant des erreurs et argumenter vos réponses.

12. Quel est généralement le rôle d'un constructeur ?
13. Qu'est-ce qu'un constructeur par défaut ?
14. Dans quelle(s) circonstance(s) le compilateur Java génère-t-il automatiquement un constructeur par défaut ?
15. Comment préciser le constructeur à appeler lors de l'instanciation d'un objet ?
16. Comment s'appelle le compilateur Java ?
17. Comment s'appelle la machine Java (interpréteur) ?

Page blanche

LES DONNEES MANIPULEES PAR JAVA

TYPES DE DONNEES MANIPULEES PAR JAVA

Java manipule deux types de données :

- Les données de type valeurs appelées *primitifs* (entiers, réels, caractères, booléens)
- Des données de type référence qui sont des télécommandes vers des d'objets.

DONNEES DE TYPE VALEUR

- Les données de type valeur (primitifs) sont allouée automatiquement en mémoire au moment de la déclaration et leur accès est très rapide.
  En l'absence d'initialisation explicite leur valeur est égale à :
  - 0 pour les **attributs d'instance ou de classe** (`false` pour les booléens),
  - quelconque pour les variables locales aux méthodes.
    Dans ce cas la donnée doit être initialisée avant utilisation sous peine d'erreur de compilation.
- Chaque type valeur possède une classe enveloppe, lui permettant d'être transformée en référence (voir plus loin).

DONNEES DE TYPE REFERENCE

- Les données de type référence (les références d'objets) sont allouées **explicitement** par l'opérateur *`new`* appliqué à une classe ou à un tableau. Nota : pour les objets de type `String` le *`new`* est implicite.
  En l'absence d'allocation explicite par l'opérateur `new`, leur valeur est égale à :
  - `null` pour les **attributs** et leur accès provoque une erreur d'exécution appelée exception.
  - quelconque pour les références locales aux méthodes. Dans ce cas la référence doit être allouée avant utilisation sous peine d'erreur de compilation.

Les opérateurs d'affection et de comparaison agissent différemment suivant que l'opération porte sur un type valeur ou un type référence. Avec les références, les opérateurs utilisent la valeur des références et non les objets référencés.

Exemple :

```
String s1 = "abc";    String s2 = s1;
if (s1==s2)  // expression incorrecte, utiliser s1.equals(s2)
```

ACCES AUX ATTRIBUTS ET METHODES (RAPPEL)

AU SEIN DE LA CLASSE

- Les méthodes d'instances ont accès sans restriction aux attributs et aux méthodes d'instance et de classe.
- Les méthodes de classe n'ont accès qu'aux attributs et aux méthodes de classe.

EN DEHORS DE LA CLASSE

- Pour les attributs d'instance leur accès n'est possible que si celui-ci est public et se fait par l'opérateur point (**.**) précédé de l'objet à utiliser :
  Exemple : `e2.toSTring();`
- Pour les attributs de classe, leur accès n'est possible que si celui-ci est public et se fait par l'opérateur point (**.**) précédé de la classe à utiliser.
  Exemple : `Etudiant.getNumbre();`

CREATION (INSTANCIATION) DES OBJETS

Un objet est créé (on dit instancié) à l'aide de l'opérateur `new`.

Le code suivant instancie trois étudiants (voir classe `Etudiant` page 4) à l'aide du constructeur spécifique : `Etudiant(`**`String, int, String`**`)` :

```
Etudiant e1 = new Etudiant("Dupond", 2010, "Eigsi");
Etudiant e2 = new Etudiant("Durant", 2011, "Centrale Nantes");
Etudiant e3 = new Etudiant("Lajoie", 2009, "Eigsi");
```

La construction suivante est interdite :

```
Etudiant e4 = new Etudiant("Lajoie", 2009); // pas de constructeur de cette signature.
```

A la suite de ces instanciations les données allouées sont les suivantes (voir définition de la classe page 4):

- 3 entiers `promo` (un par objet),
- 6 références de type `String (nom,` et `école)` (deux par objet),
- 1 entier `nombre` commun à tous les objets.

REPRESENTATION TEXTUELLE D'UN OBJET

La représentation textuelle des propriétés d'un étudiant est rendue possible grâce à la définition d'une méthode `toString` respectant la signature suivante : `public String toString()`.

Cette méthode est systématiquement appelée par la méthode `println` lorsqu'on lui passe la référence d'un objet.

```
public String toString()
{
     return nom + " : " + école + " promo " + promo;
}
```

- L'appel de la méthode suivante
  `System.out.println(e3);`
  `Affiche : Lajoie : Eigsi promo 2009`
- La majorité des classes du JDK définissent cette méthode.

DESTRUCTION DES OBJETS

Un objet est candidat à sa destruction quand :

- il sort de portée,
- une nouvelle référence lui est attribuée,
- le programme se termine.

L'objet n'est pas détruit immédiatement mais devient inaccessible. Il sera effectivement détruit lorsque le mécanisme de *garbage collector* incorporé dans la machine Java devient actif. Cela se produit quand le moteur d'exécution de Java ne détecte plus aucune activité du processeur ou lorsque la mémoire disponible devient insuffisante.

POINT D'ENTREE D'UN PROGRAMME JAVA

OU COMMENCE UN PROGRAMME JAVA ?

Pour être exécutable, une des classes doit comporter une méthode de classe (méthode `static`) portant la signature suivante :

```
public static void main(String[] args)
```

`arg` est un tableau de type **`String`** contenant les arguments de la ligne de commande.

L'extrait de programme suivant affiche le nombre et la valeur des arguments de la ligne de commande :

```
public static void main(String[] args)
{
  System.out.println("Le programme comporte : " + args.length + " arguments");
  for (int i=0; i < args.length; i++)
     System.out.println(args[i]);
}
```

EXERCICES

La classe `Test`, placée dans le même répertoire que la classe `Etudiant` contient la définition suivante :

```
class Test
{
   public static void main(String[] args)
   {
     Etudiant e1 = new Etudiant();
     Etudiant e2 = new Etudiant("Dupond", 2012, "Insa Lyon");
     Etudiant e3 = new Etudiant("Lajoie", 2012, "Eigi ");
     System.out.println(e3);
   }
}
```

1. Quel est le nom du fichier source contenant la classe Test ?
2. Donner la ligne de commande permettant de compiler les classes
3. Donner la ligne de commande permettant d'exécuter le programme.
4. Quel est le constructeur appelé par la ligne ci-dessous, comment s'appelle ce constructeur
   `Etudiant e1 = new Etudiant();`
5. Quel est le constructeur appelé par la ligne ci-dessous, comment s'appelle ce constructeur
   `Etudiant e2 = new Etudiant("Dupond", 2012, "Insa Lyon");`
6. Quel est la sortie du programme (affichage) ?
7. Après instanciation des objets e1, e2, e3, que contiennent les attributs de chaque objet.
8. La ligne ci-dessous provoque une erreur de compilation. Pourquoi ?
   `Etudiant e4 = new Etudiant("Dupond", 2012);`

ECRITURE ET EXECUTION DES PROGRAMMES JAVA

EDITION DES PROGRAMMES JAVA (RAPPEL)

Une classe doit être définie dans un fichier portant l'extension `.java` portant le même nom que la classe.

Exemple : pour la classe `Etudiant` le fichier contenant la définition de la classe doit s'appeler `Etudiant.java`

Ce fichier peut être édité avec un éditeur quelconque, mais en pratique on préfère utiliser des environnements de développement logiciels (IDE) facilitant la programmation.

Si un fichier Java contient plusieurs classes (non conseillé) une seule doit posséder l'attribut `public`.

Le compilateur Java génère toujours un fichier de *byte code* .class par classe.

COMPILATION DES PROGRAMMES JAVA (RAPPEL)

Un programme est compilé avec le compilateur `javac` fourni avec le JDK, mais en pratique on préfère utiliser des environnements de développement logiciels (IDE) facilitant le repérage des erreurs de syntaxe. Cependant ceux-ci utilisent en arrière-plan le compilateur fourni par avec le JDK.

LIGNE DE COMMANDE DE COMPILATION

Exemple le programme java `Etudiant.java` est compilé par la ligne de commande suivante :

```
Javac Etudiant.java
```

Si le programme comporte plusieurs classes dans des fichiers séparés, la ligne de commande peut :

- Spécifier chaque fichier :
  Exemple: pour un programme java comportant les fichiers : `Fichier1.java`, `Fichier2.java`, `Fichier3.java` la ligne de commande de compilation est la suivante :
  `Javac Fichier1.java Fichier2.java Fichier3.java`
- Spécifier l'ensemble des fichiers :
  `Javac *.java`

Nota : l'intérêt de la compilation à l'aide de la ligne de commande est de permettre le traitement par lots sur un grand nombre de fichiers. Cette technique est utilisée pour des programmes comportant un très grand nombre de classes.

BYTE CODE JAVA

On appelle `byte code Java` le code binaire issu de la compilation. Celui-ci réside dans des fichiers d'extension `.class` à raison d'un fichier par classe.

CHARGEMENT DES OBJETS EN MEMOIRE

La machine virtuelle Java charge les fichiers de *byte code* de façon dynamique : si un objet fait référence à un autre objet de classe différente, l'objet est chargé en mémoire en recherchant dans le répertoire courant (ou dans le répertoire précisé par la variable d'environnement CLASSPATH) un fichier `.class` portant le même nom que la classe de l'objet référencée.

Exemple : Si la classe `TestUnitaireEtudiant` contient le code suivant :

```
public class TestUnitaireEtudiant
{
    public static void main(String[] args)
    {
       Etudiant[] tabEtudiants = new Etudiant[3];

       tabEtudiants[0] = new Etudiant("Dupond", 2010, "Eigsi");
       tabEtudiants[1] = new Etudiant("Durant", 2011, "Eigsi");
       tabEtudiants[2] = new Etudiant("Lajoie", 2009, "Eigsi");

       for (int i=0; i < tabEtudiants.length; i++)
           System.out.println(tabEtudiants[i]);

        System.out.println(Etudiant.combiens()+ " instanciés.");
    }
 }
```

Le fichier de *byte code* `Etudiant.class` sera chargé en mémoire à la ligne :

```
Etudiant[] tabEtudiants = new Etudiant[3];
```

PARAMETRAGE DES CHEMINS DE COMPILATION DU JDK ET DES EXECUTABLES DU JRE

Ce paramétrage est indispensable pour lancer un programme à l'aide de la ligne de commande.

Pour lancer un programme en ligne de commande, il suffit d'invoquer la machine virtuelle Java (java) et lui passer le fichier .class comportant la fonction main. Cela implique que le programme java et que la bibliothèque soient trouvés.

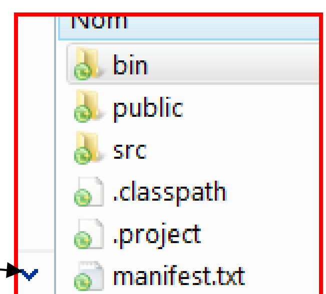
<u>La variable d'environnement PATH doit contenir le chemin vers l'interpréteur java et le compilateur javac et la variable d'environnement CLASSPATH doit contenir le chemin de la bibliothèque Java.</u> L'exemple ci-dessous est associé à la version 1.6.21 du JDK. Après installation du JDK aller dans le répertoire d'installation (généralement dans *Program Files*) et noter le répertoire d'installation du JDK et du JRE.
Nota : il faut avoir les droits administrateur pour mettre à jour les variables d'environnement.

<u>Après ces modification se déloger puis se reloger.</u>
Si vous n'avez pas la possibilité d'être administrateur sur votre machine de développement, les fichiers de commande suivants (téléchargeable sur le site http://daniel.tschirhart.free.fr/java) permettent de palier à cet inconvénient.
Ces fichiers sont à mettre à jour en fonction de l'emplacement du JRE et du JDK.

```
@Echo off
REM cjava.bat
REM Compile les programmes java présents dans le répertoire courant
REM Personnaliser éventuellement le chemin du jdk
path C:\Program Files\Java\jdk1.6.0_21\bin;%path%
javac *.java

@Echo off
REM run.bat
REM Execute le programme java spécifié par la ligne de commande
REM Personnaliser éventuellement le chemin du jre
REM Exemple d'utilisation : run mainclass
REM Avec mainclass le nom du fichier class (sans extension) comportant
REM la fonction main

java -classpath "C:\Program Files\Java\jre6\lib;." %1
```

RENDRE UN PROGRAMME JAVA EXECUTABLE : ARCHIVE JAR

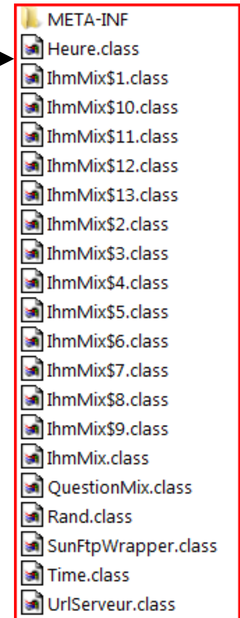
Lorsque le programme java comporte une IHM graphique et plusieurs classes (cela marche aussi avec une seule classe) il est souhaitable de créer un fichier jar. Un fichier jar est une archive en format zip comportant tous les fichiers .class de votre application plus un fichier *manifest* précisant entre autre quelle est la classe qui comporte la fonction main.
<u>L'exemple ci-dessous s'appuie sur la création du fichier mix.jar.</u>
En supposant l'organisation suivante des répertoires :



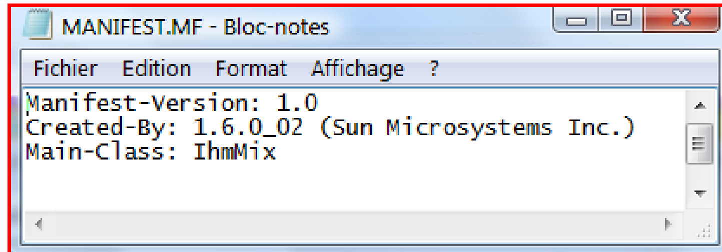
1. Créer un fichier manifest.txt (avec Notepad) à la racine du répertoire bin comportant une ligne précisant le nom de la classe contenant la fonction main.
   Par exemple pour le programme mix la classe contenant main est IhmMix :
   `Main-Class: IhmMix`
2. Créer l'archive par la commande **`jar`**.
   Procéder de façon suivante :
   Ouvrir une fenêtre de commande (exécuter `cmd.exe`) dans le répertoire `bin` contenant tous vos fichiers `.class` (vérifier par la commande **`dir`** que vous êtes dans le bon répertoire), puis taper les commandes suivantes :

```
jar -cvmf ..\manifest.txt ..\mix.jar *.class
cd ..
```

A la suite de cette commande un fichier mix.jar est créé contenant tous les fichiers .class de votre application. Ici pour l'application mix.jar on aura :



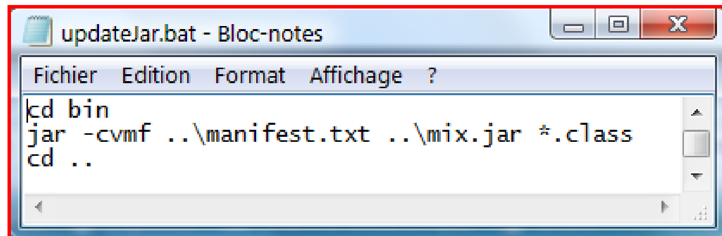
Le répertoire `META-INF` contient un fichier MANIFEST.MF spécifiant les conditions d'exécution du programme.


MANIFEST.MF - Bloc-notes

Fichier Edition Format Affichage ?

```
Manifest-Version: 1.0
Created-By: 1.6.0_02 (Sun Microsystems Inc.)
Main-Class: IhmMix
```

Pour automatiser la commande vous pouvez créer une procédure cataloguée permettant de créer le fichier jar sans retaper les lignes de commande.


updateJar.bat - Bloc-notes

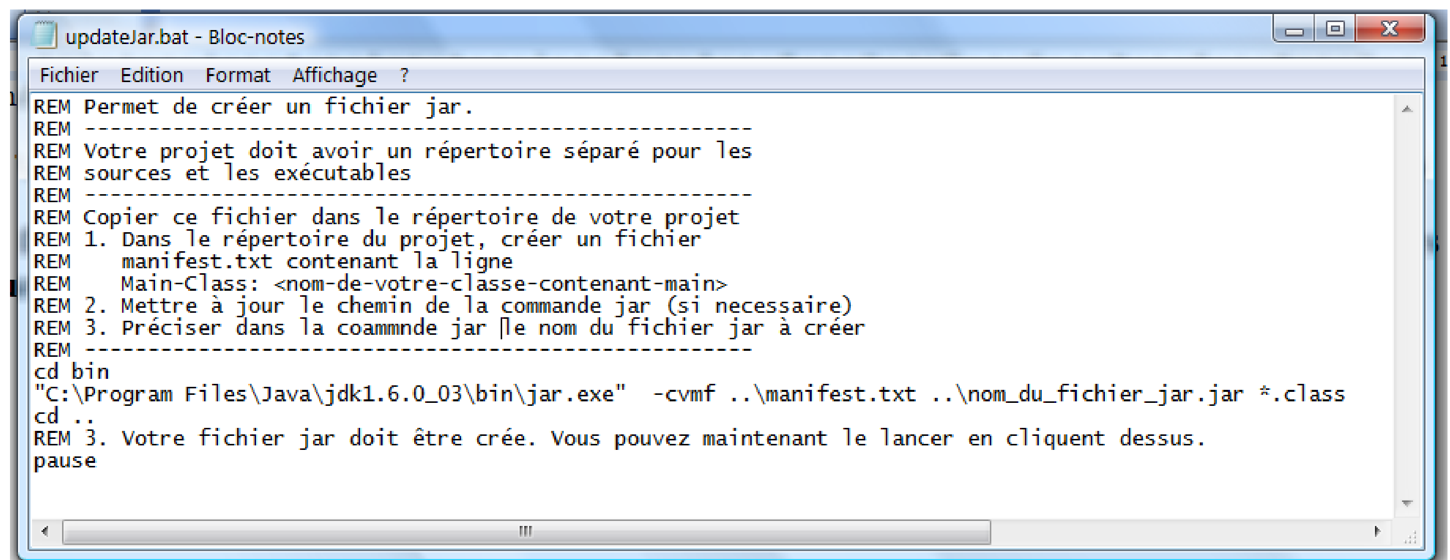
Fichier Edition Format Affichage ?

```
cd bin
jar -cvmf ..\manifest.txt ..\mix.jar *.class
cd ..
```

Si vous n'avez pas les droits administrateur sur machine de développement et que vous ne pouvez pas mettre à jour vos variables d'environnement, il faudra spécifier le chemin complet de la commande jar. Par exemple pour une machine sous Windows Seven ou XP avec le JDK 1.6.02 la commande sera :

```
"C:\Program Files\Java\jdk1.6.0_02\bin\jar" -cvmf ..\manifest.txt ..\xxx.jar *.class
```

Un fichier de commande contenant la commande updatejar.bat est disponible sur le site des cours java.


updateJar.bat - Bloc-notes

Fichier Edition Format Affichage ?

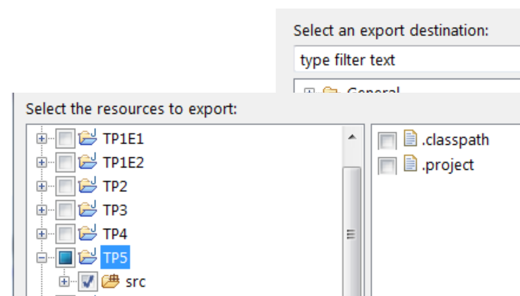
```
REM Permet de créer un fichier jar.
REM ----------------------------------------------------
REM Votre projet doit avoir un répertoire séparé pour les
REM sources et les exécutables
REM ----------------------------------------------------
REM Copier ce fichier dans le répertoire de votre projet
REM 1. Dans le répertoire du projet, créer un fichier
REM    manifest.txt contenant la ligne
REM    Main-Class: <nom-de-votre-classe-contenant-main>
REM 2. Mettre à jour le chemin de la commande jar (si necessaire)
REM 3. Préciser dans la coammnde jar le nom du fichier jar à créer
REM ----------------------------------------------------
cd bin
"C:\Program Files\Java\jdk1.6.0_03\bin\jar.exe"  -cvmf ..\manifest.txt ..\nom_du_fichier_jar.jar *.class
cd ..
REM 3. Votre fichier jar doit être crée. Vous pouvez maintenant le lancer en cliquent dessus.
pause
```

CREER UN JAR A L'AIDE D'ECLIPSE

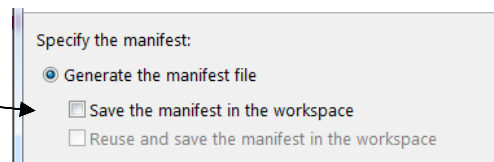
Il est possible de créer un fichier .jar directement avec Eclipse. Noter cependant que cette procédure est moins pratique surtout en phase de projet où il est nécessaire de générer souvent un fichier jar pour les tests.

L'exemple ci-dessous utilise un programme développé dans le TP *Introduction aux interfaces graphiques*.

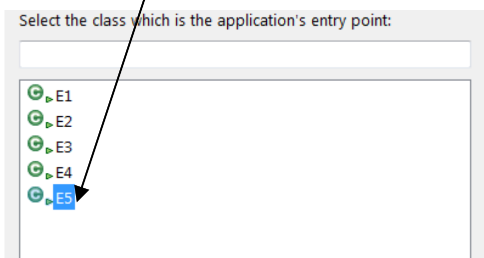
Mode opératoire :

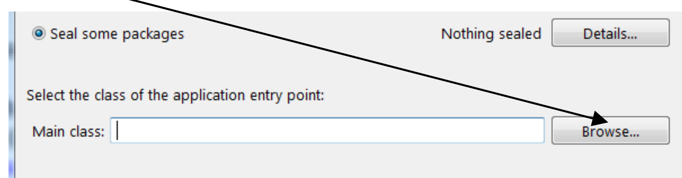


1. Sélectionner le projet dont vous voulez créer un fichier jar, puis clic droit *export* → fichier jar puis *next*.
2. Sélectionner **uniquement** le dossier src
3. Dans la même boite de dialogue, donner le chemin de destination du fichier jar, puis *next,* dans la boite de dialogue suivante accepter toutes les options par défaut puis faire *next*

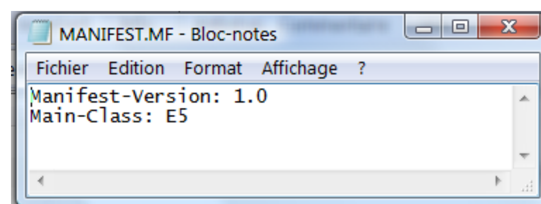


4. Cocher *Generate the manifest file*
   Utiliser *Browse* pour spécifier la classe contenant la fonction *main, ici E5* puis valider toutes les boites de dialogues.





Eclipse génère le fichier jar à l'emplacement spécifié.
Vous pouvez explorer ce fichier à l'aire d'un archiveur comme winrar ou renommer provisoirement le fichier en .zip ce qui vous permettra de l'explorer directement (sous windows).
Vérifier en particulier que le fichier `manifest` spécifie bien la classe contenant le fichier `main`.



DOCUMENTER VOS PROGRAMMES AVEC LA JAVADOC

QU'APPELLE-T-ON DOCUMENTER LES PROGRAMMES ?

Documenter les programmes ne consiste pas uniquement à commenter une classe et/ou une fonction, mais également à générer une documentation structurée partageable au sein d'un réseau Intranet ou Internet.

LA JAVADOC

Le JDK est fourni avec un outil permettant de documenter de réaliser de telles documentations. Cet outil est le programme *Javadoc* (javadoc.exe) situé dans le répertoire *bin* du JDK.

Le programme Javadoc analyse les commentaires de type /** ... */ dans les textes sources et recherche des balises commençant par @. Exemple :

```
/**
 * @author dt
 * @version 1.0
 * @docRoot Premier programme Java
 */
```

QUE PEUT-ON DOCUMENTER

Javadoc permet de documenter :

1. Un fichier Java (en dehors de la classe)
2. Une classe
3. Une méthode

Les environnements de développement comme *eclipse* facilitent grandement la mise en place des balises en particulier pour les méthodes. Les balises décrivant les paramètres formels et la valeur retournée sont automatiquement générées dès que l'on valide une ligne comportant `/**` devant la méthode.

GENERER LA DOCUMENTATION

Avec *eclipse* faire Fichier→exporter→Java→javaDoc. Préciser le chemin du générateur javadoc.exe (dans le répertoire du JDK/bin) puis valider toutes les options par défaut. JavaDoc génère alors une arborescence de site WEB dans un répertoire doc de votre projet, la racine de la documentation étant `default.html`.

EXEMPLE DE DOCUMENTATION

```
/**
 * Classe décrivant un nb complexe
 * @author dt
 * @version 1.0
 */
public class Complex
{
  private double r, i;
  /**
   * Constructeur spécifique : construit un objet à partir de re et im.
   * @param re partie réelle
   * @param im partie imaginaire
   *
   */
  public Complex(double re, double im)
  {
    r = re;
    i = im;
  }
  /**
   * Additionne deux nb complexes.<br>
   * Version statique.<br>
   * @param g opérande à gauche
   * @param d opérande à droite
   * @return g+d
   */
  public static Complex add(Complex g, Complex d)
  {
    return new Complex(g.r + d.r, g.i + d.i);
  }
  //...
}
```

Page blanche

UNE CLASSE TRES UTILISEE : LA CLASSE STRING

PRESENTATION

La classe `String` permet de décrire des chaines de caractères en format `Unicode`.
Les objets créés à partir de la classe `String` sont immuables. Cette propriété permet à un objet `String` d'être accessible de façon concurrente dans les programmes multitâches. Un objet de classe `String` peut être construit sans utiliser l'opérateur `new`, celui-ci étant utilisé de façon implicite.

Exemple :

```
1. String s = "ABC";
2. s = s + "DEF";
```

A la ligne *2*, *s* est une nouvelle instance référençant "ABCDEF". Ces lignes créées deux objets *s* dont le premier devient inaccessible à la ligne 2.
Nota : l'opérateur + est surchargé dans la classe `String` et devient un opérateur de concaténation.

CONSTRUCTEURS PRINCIPAUX

La classe String comporte 13 constructeurs ! Les constructeurs principaux sont les suivants :

Constructeur par défaut

`String();`

Exemple :

```
String s = new String();
⇔ String s = "";    et non String s; (s = null)
```

Constructeur de copie

`String(String s);`

Exemple :

```
String s = "ABC";
String s2 = new String(s);
```

Nota : du fait du caractère immuable d'un objet de type *String* ont peut écrire :

```
String s1 = "ABC";
String s2;
s2 = s1;          // s1 et s2 référencent la même instance.
s1 = s1 + "DEF";
// Nouvelle instance de s1.
// s2 pointe toujours sur l'ancienne instance de s1 c'est à
// dire "ABC"
```

Constructeurs spécifiques

`String(bytes[] by);`

Permet de construire un string à partir d'un tableau d'octets. Typiquement utilisé lors de la réception de caractères sur le Web (caractères ASCII).

Exemple

```
byte[] bTab = { 65, 66, 67 }; // 65 = caractère ascii 'A'
String s = new String(bTab);  // s  = "ABC"; (Unicode)
```

`String(bytes[] b, int offset, int length);`

Permet de construire un `String` à partir d'un tableau d'octets en commençant à l'indice offset sur une longueur *length*.

`String(char[] ch);`

Permet de construire un `String` à partir d'un tableau de caractères Unicode.

Exemple :

```
char[] cTab = { 'a', 'b', 'c' };
String s = new String(cTab);     // s  = "abc";
```

`String(char[] ch, int offset, int length);`

Permet de construire un string à partir d'un tableau de caractères en commençant à l'indice offset sur une longueur *`length`*.

`String(StringBuffer buff);`

Construit une chaine à partir d'un *`StringBuffer`*

PRINCIPALES METHODES

La classe comporte 63 méthodes !
Les principales méthodes sont les suivantes :

MANIPULATION DES CARACTERES DE LA CHAINE

`String toUpperCase();`

Converti les caractères d'une chaîne en majuscules.

Exemple :

```
String s = "abCdE";
s = s.toUpperCase();  // s = "ABCDE"
```

`String toLowerCase();`

Converti les caractères d'une chaîne en minuscules.

`String replace(char oldChar, char newChar);`

Remplace toutes les occurrences de *`oldChar`* par *`newChar`*.

Exemple:

```
String s = "(12,3 ; 30,5)";  // nombre complexe saisi
s = s.replace(',' , '.');  s = "12.3 ; 30.5)"
```

`String replaceAll(String regex, String replacement);`

Permet de remplacer les caractères décrits par une expression régulière *`regex`* par la chaîne *`replacement`*.

Exemple :

```
String s ="N°: 0033-546340012  12/8/2008";
s = s.replaceAll("\\d{4}-", "0");
// s = "N°: 0546340012  12/8/2002";
```

`String concat(String autreChaine);`

Concatène la chaine courante avec *`autreChaine`*.

Exemple :

```
String s1 = "ABC";
String s2 = "DEF");
s1 = s1.concat(s2);  // ⇔ s1 = s1 + s2
```

`String[] split(String regex);`

Découpe la chaine en tableau de `String` en se servant de la marque donnée par l'expression régulière *`regex.`*

Exemple :

```
String cplx = "(12,5 ; 50,7)";
cplx = cplx.replace('(', ' ');
cplx = cplx.replace(')' , ' ');
cplx = cplx.replace(',' , '. ');
String[] sTab = s.plit(";");
double r = Double.parseDouble(sTab[0]); // 12.5
double i = Double.parseDouble(sTab[1]); // 50.7
```

Sring trim();

Supprime de la chaîne courante les espaces en fin de chaine

Exemple :

```
String s= "123    ";
s = s.trim(); // s="123";
```

String substring(int begin);

Retourne à partir de la chaine courante une chaine débutant à la position *begin*.

Exemple :

```
String s = "CaraMiel";
System.out.println(s.substring(4)); // affiche Miel
```

String substring(int begin, int end);

Retourne à partir de la chaine courante une chaine débutant à *begin* et se terminant *end*.

Exemple :

```
String s = "Coco aime le coca";
System.out.println(s.substring(0, 4)); // affiche Coco
```

COMPARAISONS ET TESTS

int compare(String autre);

Compare la chaîne courante une chaîne avec une *autre* :

Exemple :

```
String s1="ABC";
s2 = "abc";
System.out.println(s1.compareTo(s2)); //  < 0
System.out.println(s1.compareTo(s1)); //  =0
System.out.println(s2.compareTo(s1)); // > 0
```

Nota :

```
String s3 = "ABC";
(s3==s1) vaut false (compare les références).
```

int compareToIgnoreCase(String autre);

Compare une chaîne avec une *autre* sans tenir compte de la casse.

boolean endsWith(String suffix);

Teste si la chaîne courante se termine par *suffix*.

Exemple :

```
String s1= "ABDEF";
s1.endsWith("EF"); retourne vrai
```

boolean startWith(String préfix);

Teste si la chaîne courante débute par *préfix*.

RECHERCHES

int indexOf(int ch);

Retourne la position de *ch* dans la chaîne courante ou -1. Permet donc de tester la présente d'un caractère dans une chaîne.

Exemple :

```
String s="12,3";
s.indexOf(','); // retourne 2
```

```
s.indexOf('.'); // retourne -1
```

`int indexOf(int ch, int fromIndex);`

Retourne la position de *ch* à partir de la position *fromIndex* dans la chaîne courante ou -1.

`int lastIndexOf(int ch, int fromIndex);`

Retourne dernière position de *ch* à partir de la position *fromIndex* dans la chaîne courante ou -1.

`int length();`

Retourne la longueur de la chaîne courante.

Exemple :

```
String s="12,3";
int len = s.length(); // len = 4
```

`char charAt(int index);`

Permet d'obtenir le caractère de la chaine courante à la position *index*.

Exemple :

```
String s = "+ 56";
char c = s.charAt(0); // +
```

CONVERSIONS

Les fonctions suivantes permettent de convertir en *String* des types primitifs

`String valueOf(TYPE_PRIMITIF val);`

TYPE_PRIMITIF : *char*, *int*, *long*, *float*, *double*, *boolean*

LA CLASSE StringBuffer

PRESENTATION

La classe *StringBuffer* permet de décrire des chaines de caractères en format Unicode.
Contrairement à la classe String, les objets créés à partir de la classe *String* ne sont pas immuables, c'est-à-dire peuvent être modifiés. Dans une application professionnelle, cette classe doit être utilisée à la place de la classe String lorsque des objets doivent être crées et détruits un grand nombre de fois (moins de fragmentation de la mémoire et beaucoup plus d'efficacité).

CONSTRUCTEURS PRINCIPAUX

La classe *String* comporte 4 constructeurs. Les constructeurs principaux sont les suivants :

Constructeurs

`StringBuffer();`

Construit une chaîne vide et réserve 16 caractères.

`StringBuffer(int capacity);`

Construit une chaîne vide de taille *capacity*.

`StringBuffer(String str);`

Construit une chaîne à partir d'un *string*.

PRINCIPALES METHODES

Consulter la documentation de la classe.

TRANSPYPAGE DES PRIMITIFS

AFFECTATIONS ENTRE PRIMITIFS DE TYPE DIFFERENTS

L'affectation est autorisée entre types différents, lorsqu'il n'y a pas de risque de perte d'information.
Par exemple on peut mémoriser un entier dans un réel mais pas l'inverse. Dans ce dernier cas on effectue une opération de **transtypage** explicite. Exemple :

```
int j = 10;
double d = 10;

...

d = j;          // autorisé
j = (int)d;     // transtypage explicite vers un type int
```

CONVERSION DE PRIMITIFS EN OBJET A L'AIDE DE LEUR CLASSE ENVELOPPE

Java défini une classe enveloppe pour chaque primitif. Pour les :

- entiers : classe `Integer`
- réels double précision : classe `Double`
- caractères : classe `Character`
- booléen : classe `Boolean`

NECESSITE DES CONVERSIONS

Lorsque l'on veut mémoriser un primitif dans un conteneur, il est nécessaire de convertir celui-ci en objet (les conteneurs ne contiennent que des objets). Deux techniques sont possibles.
Convertir :

- les types primitifs ⇔ en objet de classe enveloppe (opération de *boxing*)
- les types primitifs ⇔ en String

A partir de Java 5, les opérations *boxing* sont toujours présentes mais sont transparentes au programmeur.

EXEMPLE

```
public class ExempleConversions
{
  public static void main(String[] args)
  {
    // Boxing  d'un entier
    Integer objInt = new Integer(10);    // JRE > 5 Integer objInt = 10 ;

    // UnBoxing
    int i = objInt.intValue();           // JRE > 5 int i = objInt;

    // Boxing d'un réel
    Double objDouble = new Double(10.0); // JRE > 5 Double objDouble = 10.0 ;

    // UnBoxing
    double d = objDouble.doubleValue();  // JRE > 5 double d = objDouble ;

    // int -> String
    String sInt = Integer.toString(10);

    // String -> int
    int v = Integer.parseInt(sInt);

    // double -> String
    String sDouble = Double.toString(10.0);

    // String -> double
    double d = Double.parseDouble(sInt);
  }
}
```

Page blanche

FORMATAGE DES DONNEES

Le formatage de données est une opération indispensable lorsque l'on veut obtenir une présentation claire et lisible des résultats d'un traitement. On peut formater des données numériques et des chaines de caractères. Depuis Java 5, les opérations de formatage sont beaucoup plus simples, les programmeurs C retrouveront une syntaxe connue.

FORMATAGE DES FLUX

S'effectue avec la fonction `printf` implémentée dans :

- `java.io.Console`
- `java.io.PrintStream`
- `java.io.PrintWriter`

Cette fonction permet d'effectuer des sorties formatées sur l'écran, dans les fichiers et sur le réseau.
La syntaxe de la fonction : `printf(String format, Object arg1, Object arg2, ...)` est la suivante :

`Format : %[argument_index$][flag][width][.precision]conversion`

- Les `[]` indique des paramètres optionnels
- `argument_index` indique le numéro d'ordre de args (arg1 est en position 1, arg2 est en position 2 …)
- `flag` représente des caractères qui selon le format de conversion modifie format d'écriture.

Flag	Character	Floating Point	Date/Time	Description
'-'	y	y	y	The result will be left-justified.
'#'	-	y	-	The result should use a conversion-dependent alternate form
'+'	-	y	-	The result will always include a sign
' '	-	y	-	The result will include a leading space for positive values
'0'	-	y	-	The result will be zero-padded
','	-	y	-	The result will include locale-specific grouping separators
'('	-	y	-	The result will enclose negative numbers in parentheses

- `width` exprime la largeur du champ
- `precision` donne le nb de chiffres après le virgule pour les réels .
- `conversion` représente un caractère spécifiant le format de sortie. Les caractères les plus courants sont :
  - `d` pour les entiers
  - `c` pour les caractères
  - `f` pour les réels
  - `s` pour les String

FORMATAGE DES CHAINES DE CARACTERES

La fonction statique *`format`* de la classe *`String`* permet de formater les chaines de caractères avec la même syntaxe que `printf` (même paramètres, même spécificateurs de format).

EXEMPLE

```
public class Formater
{
  public static void main(String[] args)
  {
    System.out.printf("1.  **%d** \n2.  **%04d** \n3.  **%4d** \n4.  **%1$-+4d**\n", 10, 10, 10);

    System.out.printf("5.  **%f**\n", 1.0156);
    System.out.printf("6.  **%1.2f**\n", 1.0156);
    System.out.printf("7.  **%.2f**\n", 0.0156);
    System.out.printf("8   **%+.2f**\n", 0.0156);
    System.out.printf("9.  **%s**\n", "----");
    System.out.printf("10. **%6s**\n", "----");
    System.out.printf("11. **%-6s**\n", "----");
    String s="";
    s= String.format("12. **%-6s**\n", "----");
    System.out.printf(s);
  }
}
```

```
Tasks  Console
<terminated> Formatter [J
1.   **10**
2.   **0010**
3.   **  10**
4.   **+10 **
5.   **1,015600**
6.   **1,02**
7.   **0,02**
8    **+0,02**
9.   **----**
10.  **  ----**
11.  **----  **
12.  **----  **
```

Page blanche

OPERATIONS E/S CONSOLE

ENTREES DE DONNEES A PARTIR DU CLAVIER

Java ne comportant aucune méthode directe permettant de lire des données au clavier, la classe ci-dessous permet de palier à cet inconvénient.

```
import java.io.*;

public class KeyBoard
{
  public static String readString(String prompt) throws IOException
  {
    System.out.print(prompt);
    BufferedReader line_in = new BufferedReader(new
                             InputStreamReader(System.in));
    String ligne = line_in.readLine();
    return ligne;
  }

  public static int readInt(String prompt) throws IOException
  {
    System.out.print(prompt);
    BufferedReader line_in = new BufferedReader(new
                             InputStreamReader(System.in));
    int line = Integer.parseInt(line_in.readLine());
    return line;
  }

  public static double readDouble(String prompt) throws IOException
  {
    System.out.print(prompt);
    BufferedReader line_in = new BufferedReader(new
                             InputStreamReader(System.in));
    double line = Double.parseDouble(line_in.readLine());
    return line;
  }

  public static char readChar(String prompt) throws IOException
  {
    System.out.print(prompt);
    InputStreamReader isr = new InputStreamReader(System.in);
    return (char) isr.read();
  }
 }
```

EXEMPLE D'UTILISATION

```
public class Test
{
  public static void main(String[] args)
  {
    int n1 = Keyboard.readInt("Entier 1 ? ");
    int n2 = Keyboard.readInt("Entier 2 ? ");
    System.out.printf("%d * %d = %d", n1, n2, n1*n2);
  }
}
```

Page blanche

MECANISMES DE REUTILISATION

REUTILISATION !

La réutilisation de classes consiste à utiliser une classe existante dans la définition d'une nouvelle classe.
Plusieurs formes de réutilisations sont possibles en POO. Parmi les formes de réutilisations possibles on peut citer :

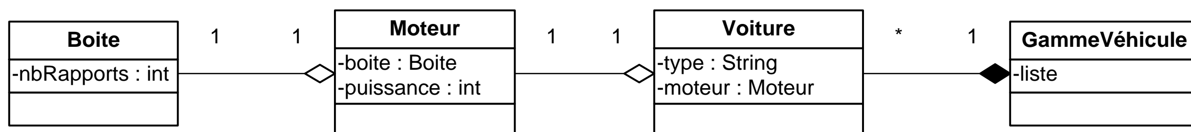
- l'incorporation d'objets dans d'autres objets sous la forme de composition ou sous la forme d'agrégation,
- l'assemblage de classes suivant un arbre hiérarchisé : l'héritage de classes

L'AGREGATION

L'agrégation est une association non symétrique, qui exprime un couplage fort et une relation de subordination.
Elle représente une relation de type "ensemble / élément".

- Une agrégation peut notamment (mais pas nécessairement) exprimer qu'une classe (un "élément") fait partie d'une autre ("l'agrégat"), qu'un changement d'état d'une classe, entraîne un changement d'état d'une autre, qu'une action sur une classe, entraîne une action sur une autre.
- A un même moment, une instance d'élément agrégé peut être liée à plusieurs instances d'autres classes (l'élément agrégé peut être partagé).
- Une instance d'élément agrégé peut exister sans agrégat (et inversement) : les cycles de vies de l'agrégat et de ses éléments agrégés peuvent être indépendants.

Exemple : Plaçons-nous du point de vue d'un constructeur gérant une gamme de véhicules.
Une gamme donnée de véhicules peut partager des éléments communs. Exemple : une berline Peugeot 206, la version coupée équivalente et la version station wagon (SW) sont équipées du même moteur et de la même boite de vitesse dans une gamme donnée. Toute modification sur le moteur ou sur la boite est répercutée sur tous les véhicules de la gamme.
Le diagramme UML traduisant ces relations est le suivant :

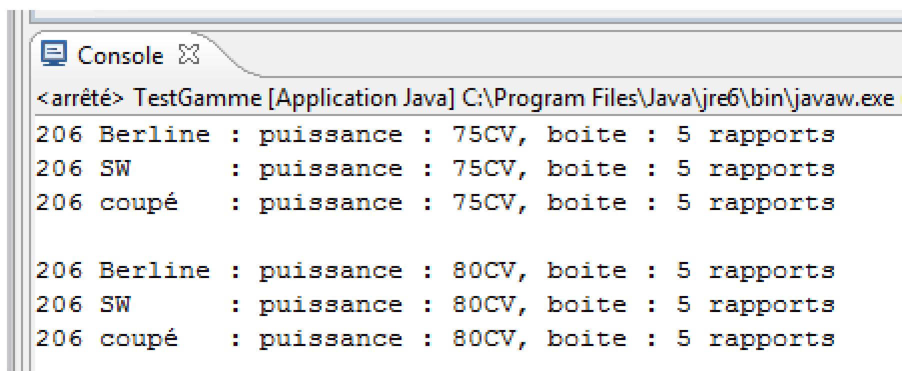


Noter ici le losange vide qui dénote l'agrégation, et le losange plein qui dénote la composition (voir plus loin) .

```java
import java.util.*;

class Boite
{
  int nbRapports;
  public Boite(int r) { nbRapports = r; }
  public String toString()
  {
    return "boite : " + nbRapports + " rapports";
  }
}

class Moteur
{
   Boite boite;
   int puissance;
   public Moteur (int p, Boite b) { puissance = p; boite = b; }
   public String toString()
   {
     return "puissance : " + puissance + "CV, " + boite;
   }
}

class Voiture
{
   String nom;
   Moteur moteur;
   // ...
   public Voiture(String n, Moteur m) {nom = n; moteur = m;}
   public String toString()
   {
     return nom + " : " + moteur;
   }
}
```

```
public class GammeVehicules
{
   private List<Voiture> gamme = new ArrayList<Voiture>();
   private Boite b1  = new Boite(5);
   private Moteur m1 = new Moteur(75, b1);
   public void ajouter(Voiture v)  { gamme.add(v); }

   public GammeVehicules()
   {
      ajouter(new Voiture("206 Berline", m1));
      ajouter(new Voiture("206 SW     ", m1));
      ajouter(new Voiture("206 coupé  ", m1));
   }

   public void modifierPuissance(int p)
   {
     m1.puissance = p;
   }

   public String toString()
   {
     String s = "";
     for (Voiture v : gamme)
       s += v + "\n";
     return s;
   }
}

public class TestGamme
{
   public static void main(String[] args)
   {
     GammeVehicules gv = new GammeVehicules();
     System.out.println(gv);
     gv.modifierPuissance(80);
     System.out.println(gv);
   }
}
```

La boite de vitesse et le moteur sont créés indépendamment d'un véhicule (`b1`, et `m1` : attributs de classe) : la suppression d'un véhicule dans la collection `gamme` n'entraine pas la suppression de la boite 5 rapports et du moteur 75CV.
Toute modification du moteur et de la boite se répercute immédiatement sur l'ensemble de la gamme de véhicules.

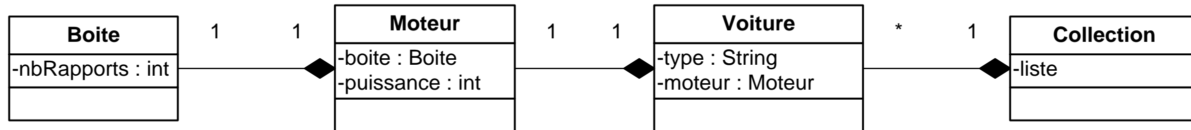


LA COMPOSITION

La composition consiste à réutiliser un objet par incorporation au sein d'une classe.

- Les cycles de vies des éléments (les "composants") et de l'agrégat sont liés : si l'agrégat est détruit (ou copié), ses composants le sont aussi.
- A un même moment, une instance de composant ne peut être liée qu'à un seul agrégat.
- Les "objets composites" sont des instances de classes composées.

Exemple : plaçons-nous du point de vue d'un collectionneur de voitures.



Chaque voiture possède un moteur et une boite spécifique.
Noter le losange plein dénotant la composition. Les classes `Boite`, `Moteur` et `Voiture` sont les mêmes que précédemment.

```
import java.util.*;

class Boite   // idem
class Moteur  // idem
class Voiture // idem

public class MaCollection
{
   public List<Voiture> gamme = new ArrayList<Voiture>();
   public void ajouter(Voiture v)  { gamme.add(v); }

   public MaCollection()
   {
     ajouter(new Voiture("Citroën 2CV  ", new Moteur(26, new Boite(4) )));
     ajouter(new Voiture("Ferrari Monza", new Moteur(450,new Boite(7) )));
     ajouter(new Voiture("Citroen DS19 ", new Moteur(90, new Boite(4) )));
   }

    public String toString()
   {
     String s = "";
     for (Voiture v : gamme)
       s += v + "\n";
     return s;
   }
}
public class TestComposition
{
  public static void main(String[] args)
  {
    MaCollection c = new MaCollection();
    System.out.println(c);
    // Supression d'un véhicule
    c.gamme.remove(0);
    System.out.println(c);
    // Nouveau véhicule
    c.ajouter(new Voiture("Citroën 2CV  ", new Moteur(29, new Boite(4) )));
    System.out.println(c);
  }
}
```

La suppression d'un véhicule dans la collection entraine la suppression de son moteur et de sa boite.
L'ajout d'une voiture dans une collection se fait en instanciant une voiture :

```
      c.ajouter(new Voiture(...)
```

Celle-ci instancie à son tour un moteur qui à son tour instancie une boite.
Contrairement à l'exemple précédent il n'y a ici pas de partage d'objet même s'il y partage d'un ensemble des définitions des classes.

```
Console
<arrêté> TestComposition [Application Java] C:\Program Files\Java\jre6\bin\javaw.exe
Citroën 2CV   : puissance : 26CV, boite : 4 rapports
Ferrari Monza : puissance : 450CV, boite : 7 rapports
Citroen DS19  : puissance : 90CV, boite : 4 rapports

Ferrari Monza : puissance : 450CV, boite : 7 rapports
Citroen DS19  : puissance : 90CV, boite : 4 rapports

Ferrari Monza : puissance : 450CV, boite : 7 rapports
Citroen DS19  : puissance : 90CV, boite : 4 rapports
Citroën 2CV   : puissance : 29CV, boite : 4 rapports
```

L'HERITAGE

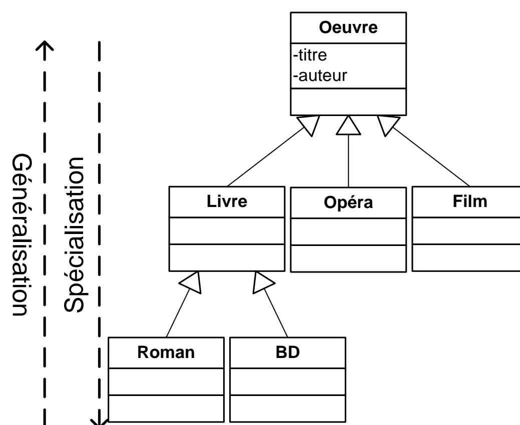
L'héritage de classe permet de construire des hiérarchies de classes a l'aide du mot clé `extends`.
Les hiérarchies de classes permettent de gérer la complexité, en ordonnant les objets au sein d'arborescences de classes, d'abstraction croissante. Deux approches sont possibles :

- La spécialisation
  - Démarche descendante, qui consiste à capturer les particularités d'un ensemble d'objets non discriminés par les classes déjà identifiées.
    On étend les propriétés d'une classe, sous forme de sous-classes, plus spécialisées (permet l'extension du modèle par réutilisation).
- Généralisation
  - Démarche ascendante, qui consiste à capturer les particularités communes d'un ensemble d'objets, issus de classes différentes.
    On factorise les propriétés d'un ensemble de classes, sous forme d'une *super-classe* plus abstraite (permet de gagner en généricité).

COMMENT FAIT-ON !

L'héritage (spécialisation et généralisation) permet la classification des objets.

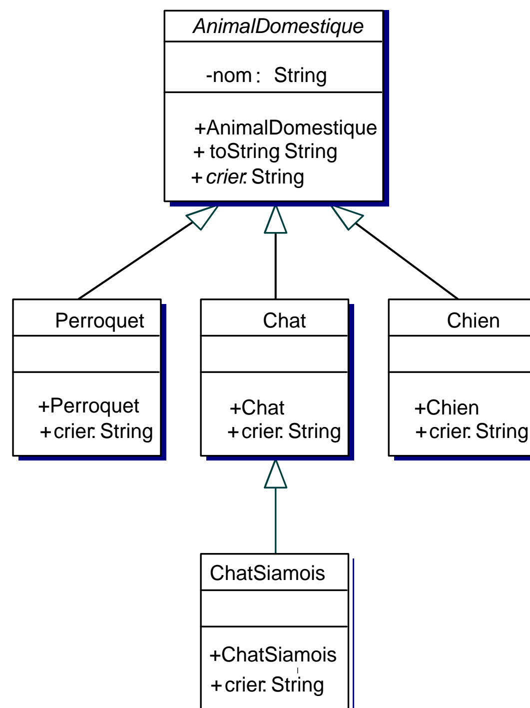
- Une bonne classification est stable et extensible : ne classifiez pas les objets selon des critères instables (selon ce qui caractérise leur état) ou trop vagues (car cela génère trop de sous-classes).
- Les critères de classification sont subjectifs.
- Le principe de substitution (Liksow, 1987) permet de déterminer si une relation d'héritage est bien employée pour la classification :
  "*Il doit être possible de substituer n'importe quel instance d'une super-classe, par n'importe quel instance d'une de ses sous-classes, sans que la sémantique d'un programme écrit dans les termes de la super-classe n'en soit affectée.*"
- Si Y hérite de X, cela signifie que "Y est une sorte de X" (analogies entre classification et la théorie des ensembles).



Exemple

Le graphe d'héritage suivant présente un exemple (très modeste) de relation entre des classes factorisant un comportement d'animaux domestiques.
Pour vérifier si l'héritage est utilisé à bon escient, on peut dire qu'un chat Siamois est une sorte de chat et le chat est une sorte d'animal domestique. Les relations d'héritage permettent de factoriser les comportements communs à toutes ces classes.
Dans la classe racine on trouvera :

- un attribut `nom` permettant de mémoriser le nom de l'animal domestique,
- une méthode `toString` permettant d'obtenir le nom de l'animal domestique,
- une méthode `crier` permettant d'obtenir le cri de l'animal domestique. Cette dernière méthode ne peut pas être définie, puisqu'elle ne correspond à aucun animal.
  Il faut redéfinir `crier` dans chaque classe.
  Exemple : la classe `AnimalDomestique` instancié ci-dessous doit se comporter de façon suivante:




```
AnimalDomestique[] maMénagerie = new AnimalDomestique [2];
maMénagerie[0] = new Chat ("Sysvestre");
maMénagerie[1] = new Chien("Médor");
System.out.println(maMénagerie[0].crier()); // doit afficher le cri du chat
System.out.println(maMénagerie[1].crier()); // doit afficher le cri du chien
```

CLASSES ABSTRAITES ET METHODES VIRTUELLES PURES

Pour résoudre le problème précédent il y a deux solutions possibles :

1. Définir la méthode `crier` dans la classe `AnimalDomestique` avec aucun code dans le corps de la méthode. `maMénagerie[0].crier())` est donc défini dans `AnimalDomestique` et il n'y a pas d'erreur de compilation +
   définir la méthode crier dans chaque classe héritant de la classe `AnimalDomestique`. Lors de l'exécution avec `maMénagerie[0].crier()` c'est la méthode `crier` de l'objet `Chat` qui sera utilisée (grâce à la résolution des liens à l'exécution entre l'appel de la fonction et la fonction appelée à l'exécution). Mais !
   - Si la classe `Chat` ne redéfini pas la méthode `crier`, celui-ci deviendra aphone ! Pour obliger les classes dérivées de la classe `AnimalDomestique` à redéfinir la méthode `crier` la solution que proposent les langages orientés objets dont Java est la suivante :

     Déclarer la méthode `crier` dans la classe `AnimalDomestique` sans corps de méthode. Une telle méthode est dite virtuelle pure (méthode sans définition). Une telle classe doit être préfixée avec le mot `abstract`. Une classe abstraite ne peut pas être instanciée.

     Si la classe `Chat` ne redéfini pas la méthode `crier`, elle hérite la méthode virtuelle pure `crier` de la classe parent et devient abstraite à son tour (donc non instanciable).

Pour traduire la relation d'héritage entre classe une classe `Y` et une classe `X` (parent de `Y`) on utilise le mot réservé `extends` de façon suivante :

```
class X
{
}

class Y extends X // Y dérive de X
{
}
```

Finalement le code de la classe `AnimalDomestique` et de la classe `Chat` est le suivant :

```
// Classe abstraite
public abstract class AnimalDomestique
{
  private String nom;
  public AnimalDomestique(String n)
  {
   nom=n;
  }
  public String toString()
  {
    return nom;
  }
  // Méthode virtuelle pure
  public  abstract String  crier();
}
```

```
class Chat extends AnimalDomestique
{
  public Chat(String n)
  {
    super(n);  // appel du constructeur du parent
  }

  public String crier()
  {
    return "Miaou";
  }
}
```

Le code complet de l'exemple est le suivant :

```
public abstract class AnimalDomestique
{
  private String nom;
  public AnimalDomestique(String n)
  {
   nom=n;
  }
  public String toString()
  {
    return nom;
  }
  public  abstract String  crier();
}

class Chien extends AnimalDomestique
{
  public Chien(String n)
  {
    super(n);
  }
  public String crier()
  {
    return "Whaaaf Whaaaf";
  }
}

class Chat extends AnimalDomestique
{
  public Chat(String n)
  {
    super(n);
  }
  public String crier()
  {
    return "Miaou";
  }
}

class ChatSiamois extends Chat
{
  public ChatSiamois(String n)
  {
    super(n);
  }
  public String crier()
  {
    return "Miaouououououououououou";
  }
}

class Perroquet extends
                     AnimalDomestique
{
  public Perroquet(String n)
  {
    super(n);
  }
  public String crier()
  {
    return "Cocoo";
  }
}
```

```
public class MaMenagerie
{
  public static void main(String[] args)
  {
    AnimalDomestique[] menagerie =
                             new AnimalDomestique[5];
    menagerie[0]= new Chien("Brutus");
    menagerie[1]= new Chat ("Sylvestre");
    menagerie[2]= new Chat ("Mobic");
    menagerie[3]= new Perroquet("Nestor");
    menagerie[4]= new ChatSiamois("Felix");

    for (int i=0; i < menagerie.length; i++)
      System.out.println(menagerie[i] + " " +
                                menagerie[i].crier());
  }
}
```

```
<terminated> InterfaceAnimalDomestique [Jav
Brutus Whaff Waaf
Sylvestre Miaou
Mobic Miaou
Nestor Cocoo
Felix Miaouououououououououou
```

Classes interface

L'obligation de redéfinir la méthode `crier` n'est cependant pas nécessaire pour les classes qui ne dérivent pas directement de la classe `AnimalDomestique` ce qui est le cas de la classe `ChatSiamois`. Les langages orientés objets proposent une autre solution pour obliger une classe à définir des méthodes : les **interfaces.**

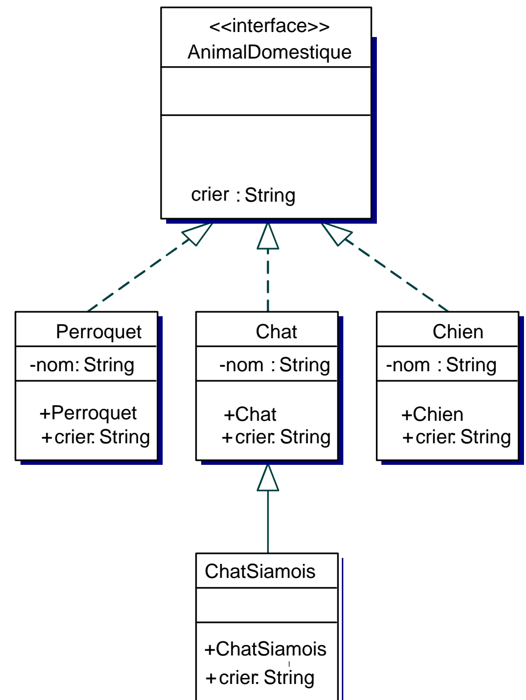
Une interface est une classe sans attributs et ne comportant que des méthodes virtuelles pures (publiques par défaut).

Si java n'autorise pas l'héritage multiple de classe, il autorise par contre l'héritage simultanées de plusieurs interfaces.

Une interface peut être vue comme un contrat passé avec une classe : la classe héritant de l'interface (on dit implémenter l'interface) doit définir toutes les méthodes de l'interface.

Exemple :

```
Public interface AnimalDomestique
{
  String crier();
}

class Chien implements AnimalDomestique
{
  private String _nom;
  public Chien(String nom) { _nom = nom;  }
  public String crier()    {return "Whaff Waaf";}
  public String toString() { return _nom; }
}

class Chat implements AnimalDomestique
{
  private String _nom;
  public Chat(String nom)  { _nom = nom;  }
  public String crier()    { return "Miaou"; }
  public String toString() { return _nom; }
}

class ChatSiamois extends Chat implements
AnimalDomestique
{
  public ChatSiamois(String n) { super(n);    }
  public String crier()    { return Miaouuoooooo";}
  public String toString()     { return _nom; }
}

class Perroquet implements AnimalDomestique
{
  private String _nom;
  public Perroquet(String nom) { _nom = nom; }
  public String crier()        { return "Cocoo";  }
  public String toString()     { return _nom;   }
}
```



La classe ma ménagerie est identique à la classe précédente.

Les exemples complets peuvent être téléchargés depuis le site http://daniel.tschirhart.free.fr/java

Page blanche

GESTION DES ERREURS D'EXECUTION : EXCEPTIONS

GERER LES ERREURS D'EXECUTION

Il est difficile, voire impossible au concepteur d'une classe, de prévoir le mode de récupération d'erreur le mieux adapté à chaque situation. C'est donc à l'application de prévoir le mode de réaction le plus approprié :

- Il faudrait que la conception de la classe incriminée laisse la possibilité de définir un mode de récupération adapté à chaque situation possible ce qui est impossible.

Face à l'impossibilité à un programme de continuer en cas d'erreur, le concepteur d'une classe peut utiliser les stratégies suivantes :

√ utiliser la politique de l'autruche !
√ terminer le programme (attention aux pertes des données !),
√ arrêter le traitement en cours et retourner une erreur dans une variable globale (mais il faut penser à la tester !)
√ appeler une fonction de gestion d'erreur qui doit toutefois se terminer avec l'une des possibilités précédentes,
√ utiliser un mécanisme à base d'exceptions.

GESTION DES ERREURS A BASE D'EXCEPTIONS

Quand l'exécution d'une application ne se déroule pas normalement, l'interpréteur java répond en levant une exception.
Par exemple si une application essaie d'ouvrir un fichier inexistant, une exception est levée.

- Lorsqu'une exception se produit, le programme Java crée un objet pour représenter cette exception.
  Java utilise une forme de gestion des exceptions structurées autour des blocs `try/catch/finally`.
  - `try` capture l'exception,
  - `catch` traite celle-ci,
  - `finaly` (facultatif) est le passage obligé qu'il y ait exception ou non.
- Si une exception n'est pas capturée par l'application, celle–ci est arrêtée par la machine Java.
- On peut créer ses propres classes d'exceptions en les dérivant la classe `Exception`.
- Les exceptions sont hiérarchisées, la dernière exception à traiter dans le bloc `catch` est l'exception de la classe `Exception`.

MISE EN PLACE D'UN MECANISME DE GESTION D'ERREURS A BASE D'EXCEPTIONS

Exemple gestion d'une division par 0.

1. Définir une classe (par exemple `ExceptionDivisionParZero`) dérivant de la classe `Exception`

```
public class ExceptionDivisionParZero extends Exception
{
  // peut être vide, java défini le constructeur par défaut
}
```

Cette classe peut cependant contenir un constructeur spécifique

```
public class ExceptionDivisionParZero extends Exception
{
  public ExceptionDivisionParZero() { }//constructeur par défaut, ne fait rien
  public ExceptionDivisionParZero(String msg) // constructeur spécifique
  {
      super(msg); // appel du constructeur Exception(String) qui mémorise le paramètre.
  }
}
```

2. Lorsqu'une division par zéro se produit, lancer l'exception (à l'aide du mot `throw`) en instanciant un objet de la classe `ExceptionDivisionParZero`.

```
static double div(double a, double b) throws ExceptionDivisionParZero
{
  if (b==0)
     throw new ExceptionDivisionParZero ();
 return a/b;
}
```

Remarquer la fonction instanciant effectuant l'opération `throw` est post fixée par :
`throws ExceptionDivisionParZero.`

`throws` esquive l'exception des objects de classe `ExceptionDivisionParZero` et oblige la fonction appelante à gérer cette exception dans un bloc catch.

On peut également utiliser le deuxième constructeur qui permettra d'afficher dans un catch la valeur des opérandes en cause :

```
static double div(double a, double b) throws ExceptionDivisionParZero
{
  if (b==0)
    throw new ExceptionDivisionParZero (String.format("%f/%f", a, b));
 return a/b;
}
public static void main(String[] args)
{
  try
  {
    for (int i=5; i >=0; i--)
      System.out.println(div(10.0,i));
  }
  catch(ExceptionDivisionParZero e)
  {
    System.out.println(e);
  }
  catch(Exception e)
  {
    System.out.println(e);
  }
}
```

ESQUIVER UNE OU PLUSIEURS EXCEPTIONS

Il est souhaitable de regrouper la gestion des exceptions en un seul endroit du programme. Il est donc indispensable de pouvoir esquiver une ou plusieurs exceptions c'est-à-dire retarder l'utilisation du bloc `try-catch`.

Dans le code ci-dessous, *pop()* lève une exception *ExceptionPileVide* et *add(..)* lève une exception *NumberFormatException.* Sans la clause `throws NumberFormatException, ExceptionPileVide`, la fonction `plus` devrait capturer ces exceptions dans un bloc `try-catch`.

La déclaration ci-dessous retarde la capture des exceptions.

```
 // Dans la classe Pile
public Complex pop()throws ExceptionPileVide
{
  if (empty()) throw new ExceptionPileVide();
  //...
}
// Dans la classe Calculette
public void plus() throws ExceptionPileVide
{
    Complex d = operand.pop();
    Complex g = operand.pop();
    operand.push(Complex.Add(g, d));
}
public void run()
{
   try
   {
    //...
    plus();  // lance les exceptions NumberFormatException et ExceptionPileVide
   }
   catch (ExceptionPileVide e)      { System.out.println(e); }
   catch (Exception e)              { System.out.println(e); }
}
```

LE LANGAGE DES EXPRESSIONS REGULIERES

Le langage des expressions régulières est spécialement conçu et optimisé pour la manipulation de texte.
Ce langage comprend deux principaux types de caractères :

- les caractères littéraux.
- les métas caractères. Caractères désignant un ensemble d'opérations.

C'est le jeu de méta caractères qui confère aux expressions régulières leur puissance de traitement.

SYNTAXE DES MOTIFS D'EXPRESSION REGULIERE

La syntaxe des motifs est très riche. Elle comporte des :

- Chaînes littérales
- Méta caractères
- Classes de caractères
- Quantificateurs
- Groupes de capture
- Frontières de recherche

LISTE DES META CARACTERES

Caractère	Description
.	Remplace tout caractère
*	Remplace une chaîne de 0, 1 ou plusieurs caractères
?	Remplace exactement un caractère
()	Groupe capturant
[]	Intervalle de caractères
{}	Quantificateur
\	Déspécialise le caractère spécial qu'il précède
^	Négation ou début de ligne
$	Fin de ligne
		Ou logique entre deux sous-motifs
+	Numérateur

EXEMPLES

`[^x]`	Tous caractères sauf *x*
`[aeiou]`	Tous caractères aeiou
`[a-z,A-Z]`	Tous caractères majuscules et minuscules
`[^aeiou]`	Tous caractères sauf aeiou

SEQUENCE D'ECHAPPEMENT

Il y a un problème lorsque que l'on cherche un caractère représentant un méta caractère comme par exemple `"^"` ou `"$"`.
Le caractère `"\"` permet de supprimer la sémantique du méta caractère.
Ainsi :

`"\^"`, `"\."`, et `"\\"` représentent les littéraux `"^"`, `"."`, et `"\"`.

Dans une chaîne de caractères Java, le caractère `"\"` est considéré comme séquence d'échappement.
Ainsi :

`"\b"` représente le caractère **b** et `"\\b"` le méta caractère `\b`

REPETITIONS

`*`	Répète un nombre de fois quelconque
`+`	Répète au moins une fois
`?`	Répète zéro ou une fois
`{n}`	Répète *n* fois
`{n,m}`	Répète entre *n* et *m* fois
`{n,}`	Répète au minimum *n* fois

FRONTIERES DE RECHERCHE

Il est souvent intéressant de forcer l'emplacement des motifs recherchés : en début de ligne, en fin de mot…
Les « spécificateurs de frontière » sont résumés dans le tableau suivant :

Spécificateur	Description
^	Début de ligne
$	Fin de ligne
\b	Extrémité de mot
\B	Extrémité d'un non mot
\A	Début de la chaîne soumise
\G	Fin de l'occurrence précédente
\Z	Fin de la chaîne soumise, à l'exclusion du caractère final
\z	Fin de la chaîne

CLASSES DE CARACTERES

Les classes de caractères permettent d'utiliser des raccourcis pour spécifier des séquences de caractères souvent utilisées.

Classe	Description
\d	Un chiffre, équivalent à : [0-9]
\D	Un non chiffre : [^0-9]
\s	Un caractère blanc : [\t\n\x0B\f\r]
\S	Un non caractère blanc : [^\s]
\w	Un caractère de mot : [a-zA-Z_0-9]
\W	Un caractère de non mot : [^\w]
.	Tout caractère

Attention : le caractère \ est un caractère spécial, il doit être déspécialisé

REGLES DE CONSTRUCTIONS DES CLASSES DE CARACTERES PERSONNALISEES

Classe	Description
[abc]	**Ensemble simple**, remplace tout caractère parmi l'un des caractères suivants : a, b et c
[^ abc]	**Négation** de l'ensemble précédent
[a-z]	**Ensemble complexe**, remplace tout caractère parmi ceux de l'alphabet naturel compris entre a et z
[a-zA-Z] [a-z[A-Z]]	**Union** d'ensembles, remplace tout caractère de l'alphabet minuscule ou majuscule
[abc&&[a-z]]	**Intersection** d'ensembles, remplace tout caractère faisant parti de l'ensemble : a, b, c et aussi de l'ensemble de a jusqu'à z (c'est-à-dire uniquement a, b et c)
[a-z&&[^ abc]]	**Soustraction** d'ensembles, remplace tout caractère de l'alphabet compris entre a et z, excepté ceux de l'intervalle suivant : a, b et c

GROUPES

Les parenthèses permettent de regrouper les expressions.
Exemple :

```
(\d{1,3}\.){3}\d{1,3}
```

L'expression précédente recherche 3 séquences de 1..3 digits terminés par un « `.` » suivi d'une autre expression de 1..3 digits. Exemple `192.168.0.1`

L'expression précédente n'est cependant pas complète puisqu'elle autorise des nombres supérieurs à 255. L'expression suivante permet de corriger ce défaut.

```
((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)
```

QUANTIFICATEURS

Quantificateurs	Description
X?	Une fois
X*	Zéro, une ou plusieurs fois
X+	Une ou plusieurs fois
X{ n}	Exactement n fois
X{ n,}	Au moins n fois
X{ n, m}	Au moins n fois et jusqu'à m fois

MISE EN ŒUVRE DES EXPRESSIONS REGULIERES DANS LA CLASSE STRING

Les expressions régulières sont mises en œuvre dans la classe String à travers les fonctions `matches`, `replaceAll`, `replaceFirst`, `split`.

Exemples :

```
// Éclate le string line en tableau de mots tokens  comportant chaque mot de l'objet  line
String line = "Blanche neige et les sept nains";
String[] tokens = line.split("\\b");

// Supprime dans le string word tous ce qui n'est pas caractère alphabétique minuscule.
word = word.replaceAll("[^a-z]", "");
```

MISE EN ŒUVRE DES EXPRESSIONS REGULIERES DANS LA CLASSE PATTERN

Deux classes permettent les recherches et le remplacement de chaines de caractères :

`Pattern`
Représentation compilée d'un motif.

`Matcher`
Moteur de recherche d'un motif dans une chaîne de caractères.

et une exception

`PatternSyntaxException`
Exception lancée lorsqu'une erreur apparaît dans la syntaxe des motifs employés.

Méthode matches( ) – Matcher

La méthode `matches()` retourne vrai (`true`) si une chaîne vérifie un motif. Cette méthode existe dans les classes `Matcher` et `Pattern`.

Objet **`Matcher`**

Syntaxe : `boolean matches() ;`

Exemple :

```
// Compilation de l'expression régulière
Pattern p = Pattern.compile("[a-z]");
// Création d'un moteur de recherche sur une chaine de caractères à partir de l'expression compilée [a-z]:
Matcher m = p.matcher
// Lancement de la recherche de toutes les occurrences dans  "abc"
boolean b = m.find();
```

Méthode split

La méthode `split()` de la classe `Pattern` permet de scinder une chaîne en plusieurs sous-chaînes grâce à un délimiteur défini dans un motif. Cette méthode est similaire à celle de la classe String.

```
String[ ] split(CharSequence input [, int limit] )
```

- Le paramètre optionnel `limit` permet de fixer le nombre maximum de sous chaînes générées. Elle retourne un tableau de `String`.

Exemple :

```
// Compilation de l'expression régulière
Pattern p = Pattern.compile(":");
// Séparation en sous-chaînes de "un:deux:trois"
String[] items = p.split("un:deux:trois");
```

Remplacements

La classe Matcher offre des fonctions qui permettent de remplacer les occurrences d'un motif par une autre chaîne.

Syntaxe :

- `String` **`replaceFirst`**`(String replacement)`
- `String` **`replaceAll`**`(String replacement)`

Ces méthodes remplacent respectivement la première occurrence et toutes les occurrences du motif de la *regex* compilée associés au moteur.

Exemple complet :

```
// Compilation de la regex avec le motif : "thé"
Pattern p = Pattern.compile("thé");
// Création du moteur associé à la regex sur la chaîne "J'aime le thé."
Matcher m = p.matcher("J'aime le thé.");
// Remplacement de toutes les occurrences de "thé" par "chocolat"
String s = m.replaceAll("chocolat");
```

Autres exemples

1. Extraire d'une chaine de caractères, une chaine commençant par une lettre minuscule ou majuscule suivi d'au moins un caractère quelconque jusqu'à la fin de la chaine.

Exemple de chaine : `String line = "01 -t2 contient 1, 2, 3"`
Valeur à extraire `"-t2 contient 1, 2, 3"`

```
Pattern ps = Pattern.compile("[a-zA-Z,-].+$");
```

`"[a-zA-Z,-].+$"` signifie caractère majuscule ou minuscule ou – suivi d'au moins 1 caractère quelconque jusqu'à la fin de la chaine.

```
match_s = ps.matcher(line);
if (match_s.find())
{
   String chaineExtraite = match_s.group();
}
```

2. Extraire une information d'une page WEB.

```
// Cet exemple extrait de la page météo :
// - http://fr.weather.yahoo.com/FRXX/FRXX0044/index_c.html
//   l'heure de coucher du soleil et le taux d'humidité à la Rochelle


import java.io.*;
import java.net.URL;
import java.util.*;
import java.util.regex.*;

public class ReadURL
{
  // Liste dans laquelle sera mémorisée la page
  List<String> liste = new ArrayList<String>();

/**
 * Charge la page dans une liste à partir de l'URL spécifiée
 * @param url : url de la page
 */
  public void readURL(String url)
  {
    try
    {
      URL webURL = new URL(url);
      BufferedReader ligne_in = new BufferedReader(new InputStreamReader(webURL.openStream()));
```

```
      String ligne;
      while ((ligne = ligne_in.readLine()) != null)
      {
        liste.add(ligne);
      }
      ligne_in.close();
    }
    catch (Exception e)
    {
      System.out.println(e);
    }
  }

  /**
   * Cherche une information spécifiée
   * @param val : paramètre spécifiant l'action (1 : lire heure de coucher, 2 ...)
   * @param liste : contient la page dans laquelle est située l'information
   * @return
   */
  public String find(int val, List<String> liste)
  {
    String ligne;
    Pattern pn;
    Matcher match_n;
    switch (val)
    {
      case 1 :
        // Exemple 1 : on cherche une ligne contenant Coucher.....XX:XX (X = digit)
        pn = Pattern.compile("Coucher.*\\d\\d:\\d\\d");
        for (String s : liste)
        {
           match_n = pn.matcher(s);
           if (match_n.find())
           {
             // On détermine la position du résultat dans la ligne
             int pos = match_n.end();
             // puis on récupère l'info entre la position -5 et 0 de la fin du motif
             String res = s.substring(pos-5, pos);
             return res;
           }
        }
      break;
      case 2 :
        // Exemple 2 : on spécifie directement la chaine avant l'info recherchée
        pn = Pattern.compile("Humidité:</font></td><td><font size=-1>");
        for (String s : liste)
        {
           match_n = pn.matcher(s);
           if (match_n.find())
           {
             int pos = match_n.end();
             // puis on récupère l'info entre la fin du motif position 0 et la position +3
             String res = s.substring(pos, pos+3);
             return res;
           }
        }
      break;
     default: return null;
    }
    return null;
  }

  public static void main(String[] args)
  {
    ReadURL rUrl = new ReadURL();

    rUrl.readURL("http://fr.weather.yahoo.com/FRXX/FRXX0044/index_c.html");
    System.out.println("Heure de coucher du soleil à La Rochelle : " + rUrl.find(1, rUrl.liste));
    System.out.println("Taux d'humidité  actuel    à La Rochelle : " + rUrl.find(2, rUrl.liste));
  }
}
```

Page blanche

INTRODUCTIONS AUX COLLECTIONS (CONTENEURS)

TYPES DE COLLECTION

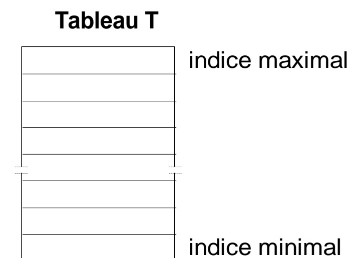
Les collections ou Type de Données Abstrait (TDA) se rangent en deux catégories

1. Collections séquentielles : les données sont enregistrées de façon séquentielle.
    - Vecteurs, piles, files, sacs, ensembles.
2. Collections associatives : les données sont associées à une clef.
    - Dictionnaire : `Map`

COLLECTIONS SEQUENTIELLES

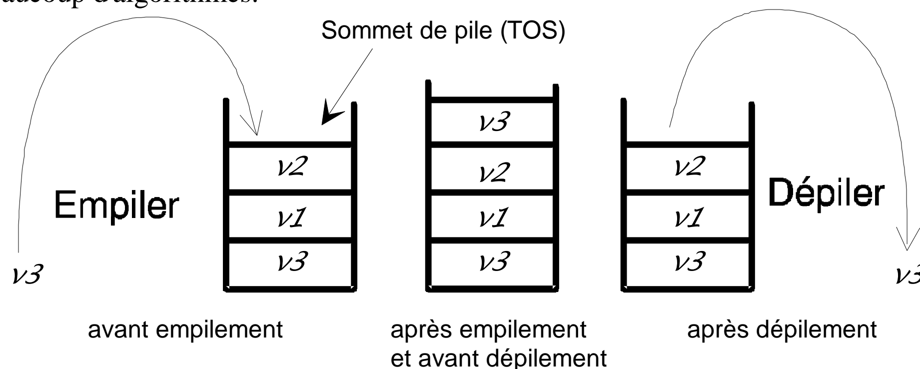
VECTEURS (ARRAY)

Dans un vecteur la donnée est directement accessible à l'aide d'un index appelé indice. Une implémentation directe d'un vecteur peut être réalisée à l'aide d'un tableau, mais d'autres structures de données sont possibles. Un vecteur se prête mal pour l'ajout ou la suppression des données.



PILE

Une pile est une structure de données à accès restreint. Deux opérations principales sont permises : *empiler* (insérer à la tête ou un sommet) et *dépiler* (supprimer au sommet).
Une pile se comporte un peu comme le dessus d'un bureau : le travail à faire s'y empile et s'y dépile régulièrement. Tous les travaux sont effectués que lorsque la pile de travail est complètement vidée. Un programme est parfois organisé de cette manière, remettant certaines tâches plus tard pour en effectuer d'autres. Les piles représentent une structure fondamentale pour beaucoup d'algorithmes.

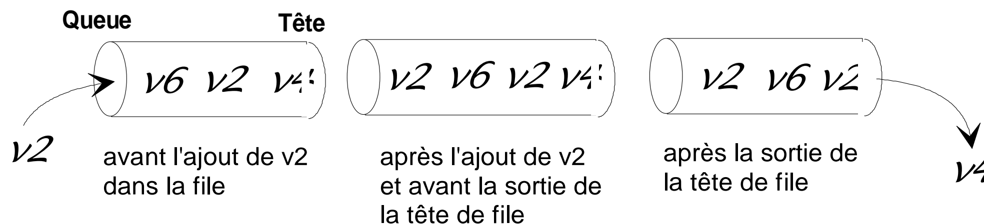


Un exemple d'utilisation des piles est l'évaluation des expressions arithmétiques :
`5 * (((9 + 8)*(4*6))+7)`
La pile constitue un moyen idéal pour mémoriser les résultats intermédiaires d'un calcul. L'expression précédente peut être calculée par :

```
empiler(5)
empiler(9)
empiler(8)
empiler(dépiler()+dépiler())
empiler(4)
empiler(6)
empiler(dépiler()*dépiler()
empiler(dépiler()*dépiler()
empiler(7)
empiler(dépiler()+dépiler())
empiler(dépiler()*dépiler())
afficher dépiler()
```

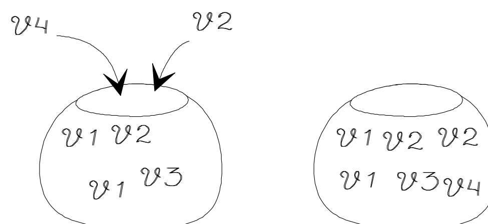
FILE FIFO

Une file est, comme la pile, une structure de données à accès restreint. Deux opérations principales sont permises : *ajouter* (insérer à la queue de la file) et *suivant* (obtenir l'élément en tête le retirer de la file). Une file *fifo* se comporte comme une file d'attente devant un guichet : les personnes arrivant se placent en queue de file, celles qui quittent la file le font à partir de la tête.
Les files représentent une structure fondamentale pour beaucoup d'algorithmes.



SACS (BAGS)

Un TDA de type sac est une collection de données non ordonnée. Un sac peut contenir plusieurs fois la même donnée. Un sac est utilisé pour stocker des objets, l'ordre dans le quel les objets sont stockés n'ayant aucune importance.



Ajout de V4 et V2 dans un sac

ENSEMBLES (SET)

Les ensembles sont similaires aux sacs, excepté qu'ils ne peuvent contenir plusieurs données identiques. Les ensembles autorisent des opérations ensemblistes.

COLLECTIONS ASSOCIATIVES

La donnée mémorisée dans la collection associative est constitué d'une paire : la clé et la donnée associée à la clé.
La clé identifie la donnée il en résulte que les clés ne peuvent être dupliquées.
On accède à la donnée de manière similaire à un tableau indicé par la clé.

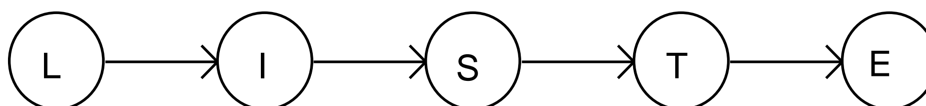
STRUCTURES DE DONNEES UTILISEES POUR FABRIQUER LES COLLECTIONS (STRUCTURES DE DONNEES FONDAMENTALES)

TABLEAU

Un tableau peut être utilisé pour implémenter directement un vecteur, une pile, une file, un sac. L'accès est très rapide mais l'ajout ou la suppression des données en dehors du haut du tableau est proportionnelle à la taille du tableau.

LISTES LIEES SIMPLES

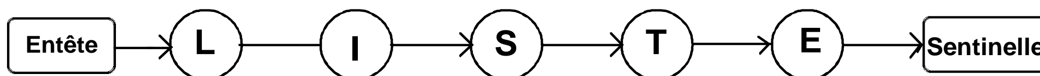
Une liste liée est un ensemble d'éléments organisés séquentiellement comme dans un tableau. Dans un tableau l'organisation séquentielle est donnée de façon implicite (par la position dans le tableau); dans une liste la liaison entre éléments est réalisée de façon explicite. Cette organisation implique que la donnée soit accompagnée d'un lien vers la donnée suivante. L'ensemble de ces informations constitue un nœud. La figure ci-dessous montre une liste liée dans laquelle les éléments sont représentés par des lettres, les nœuds par des cercles et les liens par des arcs orientés.



Liste liée.

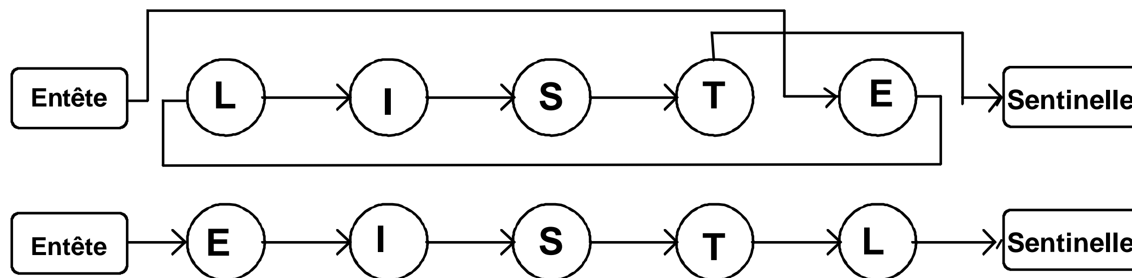
La figure met en évidence les éléments suivants :

- Chaque nœud possède un lien, ce qui implique que le dernier nœud doit référer un nœud suivant. On convient alors d'un nœud sentinelle ou nœud factice (figure ci-dessous).
- Chaque nœud est repéré par un lien précédent ce qui implique que le premier nœud doit posséder un élément qui le repère. Ce nœud sera appelé entête (figure ci-dessous). De même que la fin de la liste doit spécifier un repère de fin (sentinelle).



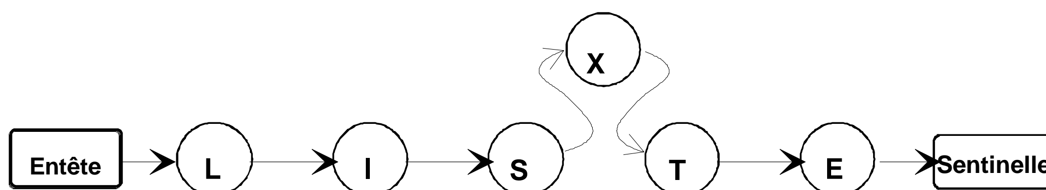
Liste liée avec sentinelles.

Grâce à cette représentation explicite de l'ordre, certaines opérations peuvent être exécutées plus efficacement que ne le permettrait un tableau. Supposons par exemple que l'on désire déplacer la lettre E de la position finale vers la première position. Dans un tableau il faudrait déplacer chaque élément pour pouvoir faire de la place en tête. Dans une liste il suffit de transformer trois liens comme indiqué dans les figures ci dessous.

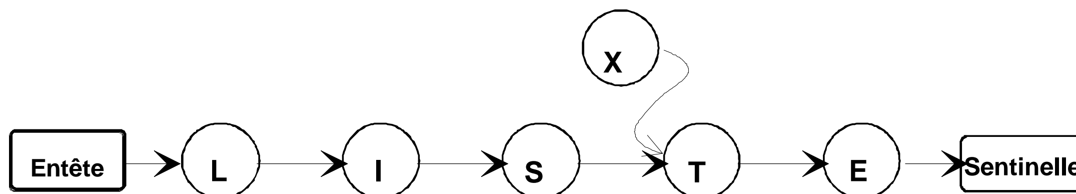


Réorganisation d'une liste.

Quel que soit la taille de la liste, cette opération ne nécessite que trois modifications dans la liste. L'insertion et la suppression d'un élément est très simple avec une liste et s'effectue avec un temps constant. Elle est par contre très mal commode avec un tableau et s'effectue avec un temps proportionnel à la taille du tableau.
L'insertion ou la suppression de la lettre X est donnée à la figure ci-dessous.



Insertion d'un élément dans une liste liée

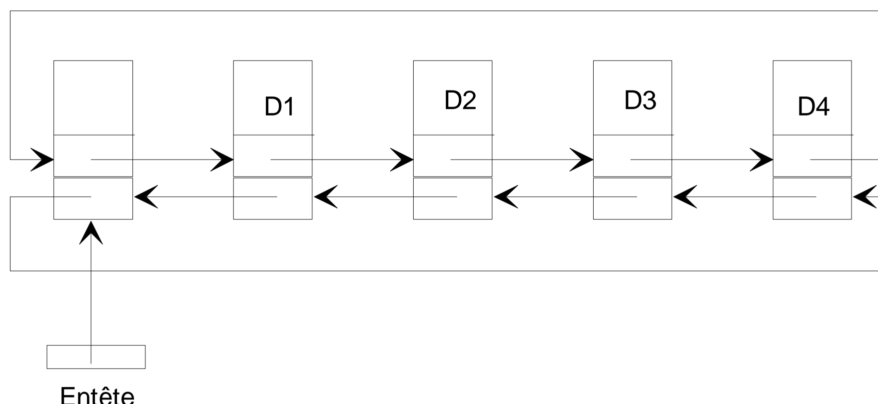


Suppression d'un élément dans une liste liée

On supprime l'élément X de la liste en faisant pointer le nœud S vers le nœud T. Le nœud X pointe toujours sur T, mais plus rien référence le nœud X.
Les listes sont mal adaptées à certaines opérations. La plus évidente est l'extraction du nième élément (trouver l'élément dont on connaît l'indice). Une autre opération est la « rechercher de l'élément précédant un élément donné ». Cette opération doit être réalisée lorsque l'on supprime un nœud. Le moyen de contourner cette difficulté consiste en une "suppression du nœud suivant".

LISTES DOUBLEMENT LIEES

Ce type de liste permet d'atteindre avec la même efficacité, la tête de liste et la fin de liste.

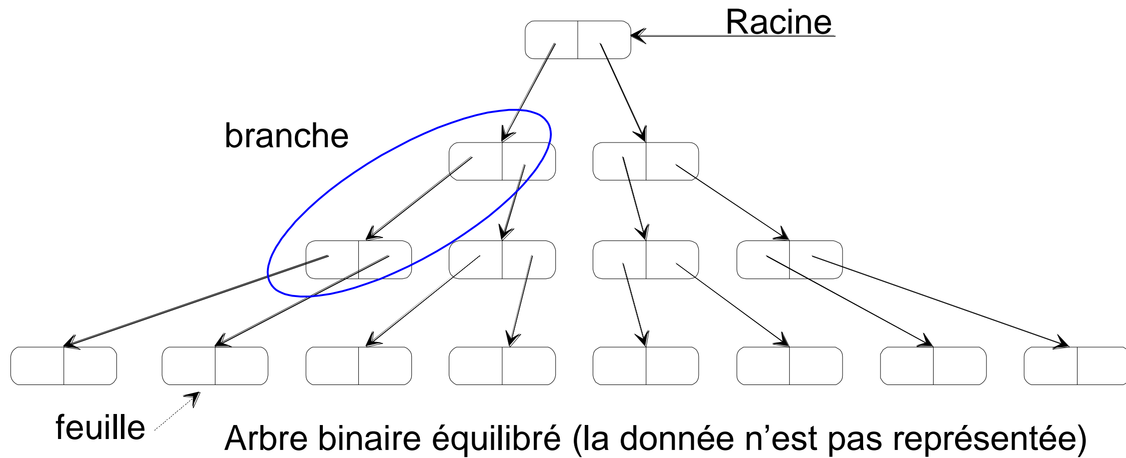


Le principe est sensiblement identique à la liste liée simple. Le nœud comporte ici, outre la donnée, deux références : une vers le nœud suivant et une vers le nœud précédent. Le premier nœud est fictif pour avoir des algorithmes réguliers.

<u>Quel que soit la structure de donnée fondamentale utilisée, le nombre de comparaisons moyen a effectuer avant de trouver une donnée parmi n est de n/2.</u>

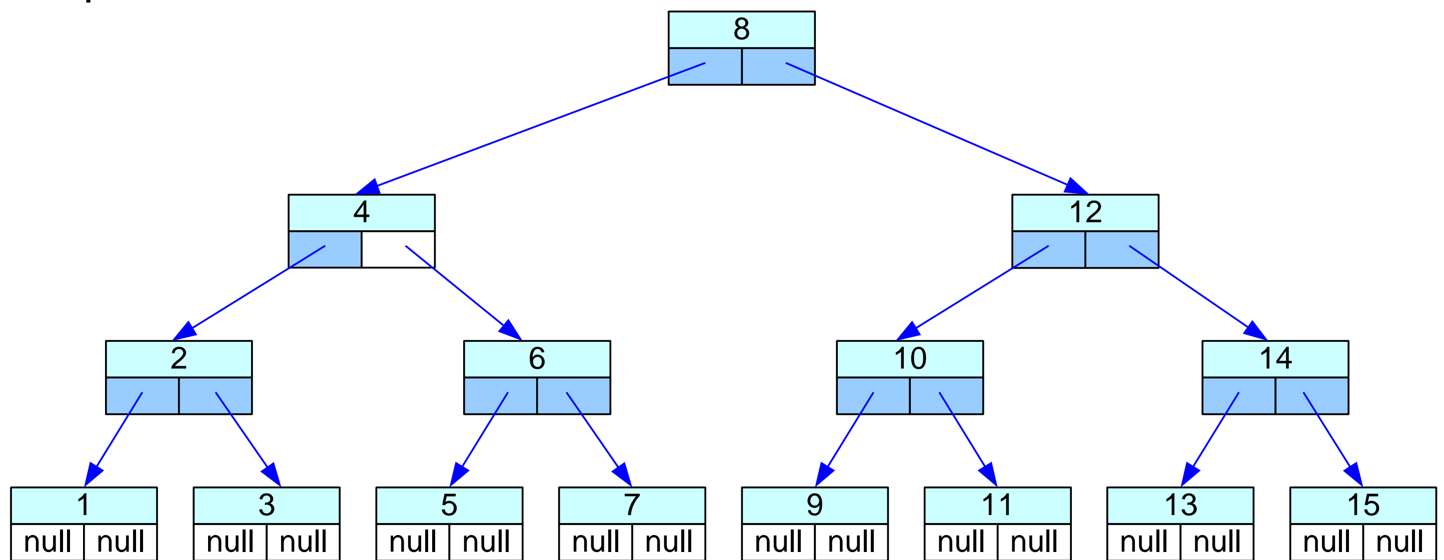
ARBRES BINAIRE

Un arbre est une liste liée où chaque nœud pointe en avant sur plusieurs nœuds.
La racine de l'arbre est le premier nœud.
Un sous-arbre est constitué d'un nœud est de tous les nœuds en dessous de lui.
Le nœud terminal (feuille) est un nœud sans descendant.
Un nœud donné a un parent (le nœud au-dessus de lui) sauf si c'est la racine, peut avoir des frères (les nœuds issus d'un même parent) ou des descendants (nœuds qu'il pointe directement). Les données sont placées dans l'arbre suivant des relations d'ordre.
Lorsque les nœuds terminaux (feuilles) sont au même niveau, l'arbre est équilibré.



Arbre binaire équilibré (la donnée n'est pas représentée)

Exemple d'arbre binaire



Arbre équilibré mémorisant les entiers de 1..15

Les données situées dans l'arbre sont naturellement triées.
Pour un arbre équilibré, le nombre de comparaisons maximale a effectuer avant de trouver une donnée parmi n est de : $\frac{Ln(n)}{Ln(2)}$

TABLES A ADRESSAGE DISPERSE (HASTABLE)

Principe d'une table de hachage

Une table de hachage calcule un nombre entier appelé code de hachage pour chacun des éléments à mémoriser (ce calcul n'est effectué qu'une seule fois). Ce code qui doit être le plus unique possible sert d'indice pour placer la données (sceau) dans un tableau.
Pour trouver la place d'un élément dans le tableau (numéro du seau) on calcule son code de hachage puis on le réduit par le modulo du nombre total de seaux. Exemple :
Code de hachage calculé = 345 avec 100 seaux, l'objet est placé dans le seau 45 (345 modulo 100=45).
Avec un peu de chance le seau sera vide, sinon (collision de hachage) on place la donnée dans une liste.
En utilisant des codes de hachage distribués aléatoirement et avec un nombre de seaux suffisamment grand (+50% de la taille de collection) on aura en général des listes courtes.

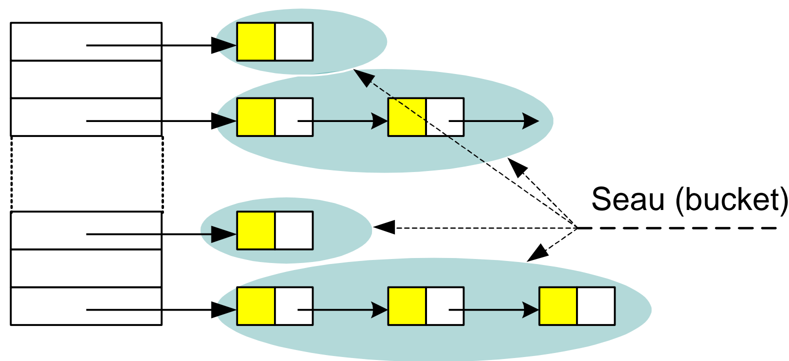


Table de hachage

- La table de hachage est constituée d'un tableau de listes chaînées
- Chaque emplacement de tableau est appelé seau

Les tables à adressage dispersé ou table de hachages permettent d'accélérer (considérablement) la recherche d'un élément dans un conteneur non trié.
Exemple : sans hachage la recherche d'un élément dans un conteneur séquentiel nécessite en moyenne de parcourir taille/2 éléments (taille = nombre d'éléments dans le conteneur), avec une table de hachage la donnée est trouvée soit immédiatement si le sceau est occupé par une liste comportant un seul élément ou au maximum avec quelques comparaisons. Les tables de hachage sont environ 30% plus rapides que les arbres binaires, mais les données ne sont pas triées.

TYPE DE DONNEES ABSTRAIT ET STRUCTURES DE DONNEES FONDAMENTALES

Les collections séquentielles sont implémentées au choix du programmeur à l'aide des structures de données fondamentales de type tableau ou liste chainée sauf pour les ensembles qui utilisent généralement des arbres ou des tableaux d'adressage dispersés.
Les collections associatives de type dictionnaire sont implémentées au choix du programmeur à l'aide des structures de données fondamentales arbres ou tableaux d'adressage dispersés.
Le choix de la structure de donnée se fait au moment de l'instanciation des collections et est ensuite transparent ce qui autorise par la suite le changement de structure de donnée sans modification du programme autre que la modification de l'instanciation. Cette modification est généralement liée par dés considérations de performances ou d'occupation mémoire, il est parfois difficile de déterminer par avance la structure de donnée la plus adaptée.

Page blanche

LES CLASSES PARAMETREES

Nécessité de transtyper les types *Object*

Soit une pile d'éléments définie par une classe *Pile.*

```
public class Pile
{
  private Object[] objects;
  ...
  public Pile()   {  }
  // Ajouter un élément dans la Pile
  public void empiler(Object object)
  {
    ...
  }
  // Obtenir et retirer l'objet situé au sommet Pile
  public Object depiler()
  {
    ...
  }
}
```

L'instanciation d'un objet `p` de la classe `Pile` s'écrit

```
Pile p = new Pile ();
```

Comme la pile peut contenir n'importe que type (la classe *Object* est la classe parente de toutes les classes)
On peut écrire :

```
p.empiler("Dupond");
p.empiler(new Double(5.0)); // Boxing d'un double
p.empiler(new Pile());
p.empiler(new Complex());
p.empiler(5);  // erreur 5 n'est pas un objet
```

Lorsqu'on effectue un dépilement, le compilateur Java ne peut pas connaitre le type réel à dépiler, il a besoin du programmeur pour lui indiquer quel type de donnée dépiler. **Cette opération appelée transtypage** est très risquée, car si le type ne correspond pas, c'est le *crash* assuré.

```
String s = (String) p.depiler();               // OK si la pile contient un String
int v = ((Integer) p.depiler()).intValue(); // OK si la pile contient un Integer
```

Le dernier exemple illustre également la nécessité d'effectuer une opération *unboxing* pour obtenir une valeur numérique.

UTILISATION DES CLASSES GENERIQUES

La généricité permet de paramétrer les classes avec des types, c'est-à-dire que le compilateur connait le type réel des objets utilisés. Pour la classe `Pile` on écrit

```
Pile<Complex> p1 = new Pile<Complex>(); // Pile de Complex
p.push(new Complex());   // OK
p.push(new Double(5.0)); // erreur de compilation
p.push(5);               // erreur de compilation
```

ATTENTION : le paramétrage ne concerne que les objets et non les primitifs

```
Pile<int>     p1 = new Pile<int>();          // interdit
Pile<Integer> p2 = new Pile<Integer>();      // correct
```

(La première ligne est barrée dans l'original.)

BOXING ET UNBOXING AUTOMATIQUE

Rappel : une opération de *boxing* est une opération permettant de transformer un primitif (type valeur) en objet grâce à sa classe enveloppe. Une opération de *unboxing* est une opération permettant de transformer un objet encapsulant un type primitif dans une classe enveloppe.

Avec la définition :

```
Pile p = new Pile();
```

On doit écrire :

```
    p.push(new Integer(5));                     // boxing
    int j = ((Integer)p.depiler()).intValue();  // unboxing
```

Avec la définition :

```
Pile<Integer> p= new Pile<Integer>();
```

On peut écrire :

```
    p.push(5); // boxing automatique
```

L'expression précédente est traduite en :

```
    p.push(new Integer(5));
```

De même

```
    int val = p.pop(); // unboxing automatique
```

est traduit en :

```
    int val = ((Integer)p.depiler()).intValue();
```

DEFINITION D'UNE CLASSE GENERIQUE

Exemple : Classe `Pile`

```
public class Pile<T>
{
  private Object[] objects;
  // et non  private T[] objects;
  // un seul fichier Pile.class quel que soit le type

  public Pile()   { ... }

  public void push(T object)
  {
    ...
  }

  public T pop()
  {
    ...
  }
}
```

INSTANCIATION D'UNE CLASSE GENERIQUE

Les classes génériques comportent de nombreuses limitations contrairement au C++, elles ne peuvent utiliser dans leur paramétrage que des classes et non des types primitifs.
On peut écrire :

```
      Pile<Integer> p= new Pile<Integer>();
```

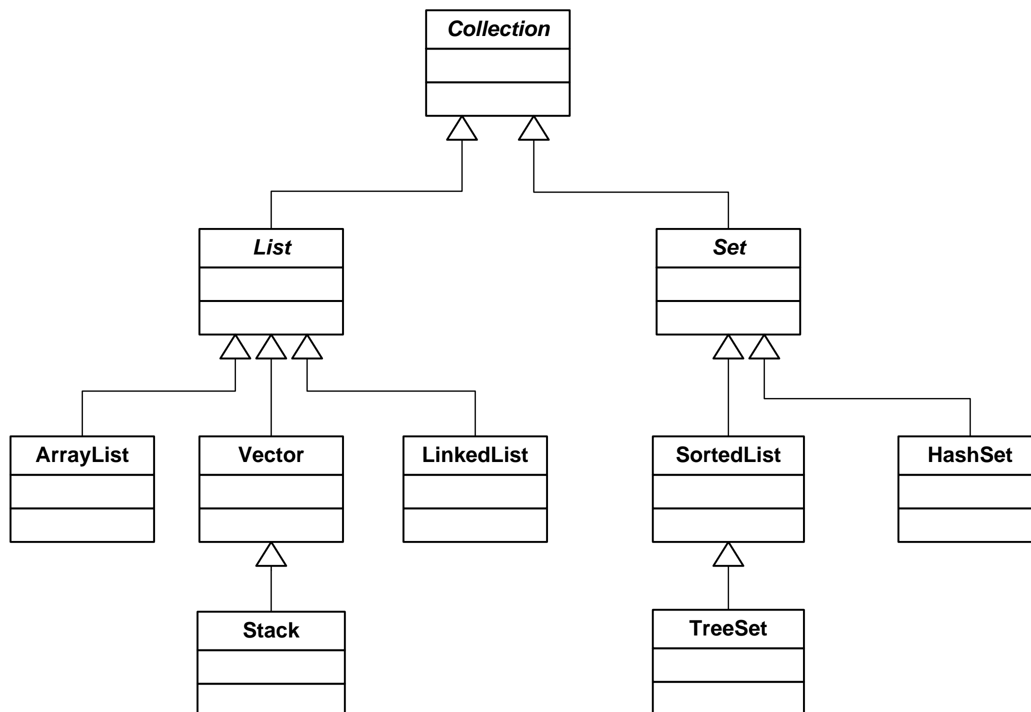
Mais pas :

~~`Pile<int> p= new Pile<int>();`~~

LES COLLECTIONS SEQUENTIELLES DANS JAVA

GRAPHE D'HERITAGE SIMPLIFIE DES COLLECTIONS SEQUENTIELLE JAVA

La figure ci-dessous présente une vue **très simplifiée** (voire partiellement incorrecte) du graphe d'héritage des collections séquentielles utilisées dans Java.
Toutes les collections sont paramétrées.



La classe parente à tous les conteneurs séquentiel est la classe abstraite *Collection*.
Cette classe possède de nombreuses fonctions permettant de rechercher le maximum ou le minium d'une collection, de trier les collections etc, … Chacune de ces méthodes possède deux formes :
Si les éléments peuvent directement être comparés, les éléments sont transmis directement, dans le cas contraire, un comparateur doit être passé dans le deuxième paramètre. Dans ce cas, la donnée doit implémenter l'interface *Comparable* et définir les méthodes de l'interface soit *compare* et *equal*

PARCOURT DES COLLECTIONS

Tous les conteneurs sauf `Stack`, sont parcourus de la même manière à l'aide d'un itérateur.
Un iterateur est un objet permettant de faire l'abstraction de la structure de donnée fondamentale sous-jacente du conteneur. Ainsi les conteneurs `Vector`, `ArrayList`, `LinkedList` incorporent tous un objet de type `Iterator`.

Principales méthodes d'un itérateur Java

`boolean`	**`hasNext()`**	Returns `true` if the iteration has more elements.
<u>**T**</u>	**`next()`**	Returns the next element in the iteration.
`void`	**`remove()`**	Removes the last element returned by the iterator

Exemple de parcours d'une liste

```
List<Complex> l = new LinkedList<Complex>();
l.add(new Complex(3,5));

Iterator<Complex> it = l.iterator();
while (it.hasNext())
  System.out.println(it.next());
```

Java 5 possède une nouvelle construction for (`foreach`) simplifiant le parcours d'une collection.

```
for (String s : l1)
  System.out.println(s);
```

Pour supprimer un élément du conteneur il faut l'avoir lu auparavant :
Exemple :

```
it.next();
it.remove();
it.remove();   // faux

it.next();     // correct
it.remove();
it.next();
it.remove();
```

INTERFACE LIST (T.D.A.)

Cette interface déclare les méthodes que doit implémenter `ArrayList`, `Vector`, `LinkedList`, ce qui implique que le code d'une application utilisant une liste `List` (le TDA) implémentée avec une `ArrayList` (c'est la S.D.F) est totalement interchangeable avec une `List` implémentée avec un `Vector` ou une `LinkedList`

boolean	**add**(T o) ajoute en fin de liste
T	**get**(int index)
void	**clear**() supprime tous les éléments
boolean	**contains**(Object o)
boolean	**equals**(Object o)
boolean	**isEmpty**()
Iterator<T>	**iterator**() Retourne un itérateur
boolean	**remove**(Object o)
int	**size**()

CLASSES CONCRETES VECTOR, ARRAYLIST, LINKEDLIST (S.D.F.)

Ces conteneurs sont des tableaux dynamiques (leur taille augmente automatiquement). Ils sont bien adaptés pour manipuler les données situés à un indice quelconque, mais sont mal adaptés lorsque l'on ne connait pas par avance le nombre de données à mémoriser (agrandissement dynamique de la taille du tableau) auquel cas il faut leur préférer la liste liée *LinkedList*. Toutes ces classes dérivent de l'interface *List.*

- Classe `Vector`
  Tableau qui s'agrandi automatiquement. La classe est dite synchronisée : l'accès d'un objet de type `Vector` peut être effectué de façon concurrente. Exemple d'utilisation :
  ```
  List<Complex> l = new Vector<Complex>();
  l.add(new Complex(4, 4));
  ```

- Classe `ArrayList`
  Similaire à `Vector` sauf que la classe n'est pas synchronisée.
  Exemple d'utilisation :
  ```
  List<Complex> l = new ArrayList<Complex>();
  l.add(new Complex());
  ```

Pour ces classes, la durée de suppression ou d'insertion d'un objet au milieu de la collection est proportion à la taille de la collection. Le tri est par contre très rapide.

- Classe `LinkedList`
  Liste d'éléments implémentés à l'aide d'une liste chainée. La classe n'est pas synchronisée.

Exemple :

```
import java.util.*;
public class TestListe1
{
  public static void main(String[] args)
  {
    List<String> l1 = new ArrayList<String>();
    l1.add("-L3-"); l1.add("-L2-"); l1.add("-L1-"); l1.add("-L0-");
    // Afficher la liste
    System.out.println(l1);
    // Parcourir la liste
    for (Iterator<String> it = l1.iterator(); it.hasNext(); )
      System.out.println(it.next());
    // Trier la liste
    Collections.sort(l1);
    System.out.println(l1);
    // Parcourir la liste avec foreach
    for (String s : l1)
      System.out.println(s);
    //Supprimer le deuxième élément;
    Iterator<String> it = l1.iterator();
    it.next();
    it.next();
    it.remove();
    System.out.println(l1);
  }
}
```

Sortie

```
[-L3-, -L2-, -L1-, -L0-]
-L3-
-L2-
-L1-
-L0-
[-L0-, -L1-, -L2-, -L3-]
-L0-
-L1-
-L2-
-L3-
[-L0-, -L2-, -L3-]
```

LES ENSEMBLES : INTERFACE SET (T.D.A.)

Cette interface déclare les méthodes que doit implémenter `HashSet` et `TreeSet`, ce qui implique que le code d'une application utilisant un ensemble `Set` (le TDA) implémenté avec une `HashSet` (c'est la S.D.F) est totalement interchangeable avec un ensemble `Set` implémenté avec un `TreeSet`.

boolean	**add**(E e) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(Object o) Returns `true` if this set contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns `true` if this set contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this set for equality.
Int	**hashCode**() Returns the hash code value for this set.
Boolean	**isEmpty**() Returns `true` if this set contains no elements.

Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
Boolean	**remove**(Object o) Removes the specified element from this set if it is present (optional operation).
Boolean	**removeAll**(Collection<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
Boolean	**retainAll**(Collection<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
Int	**size**() Returns the number of elements in this set (its cardinality).
Object[]	**toArray**() Returns an array containing all of the elements in this set.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.

CLASSE CONCRÈTES HASHSET, TREESET (S.D.F.)

HashSet et *TreeSet* sont des classes implémentant l'interface *Set.*
HashSet utilise une table d'adressage dispersée, alors que *TreeSet* utilise un arbre binaire.
L'accès aux éléments d'un ensemble ne se fait pas à l'aide d'un indice (mais on peut transformer l'ensemble en tableau) et une seule occurrence de donnée est autorisée.

Exemple

```
import java.util.*;
public class TestSet
{
  public static void main(String[] args)
  {
    System.out.println("TreeSet");
    Set<String> ts = new TreeSet<String>();
    ts.add("-1-");
    ts.add("-3-");
    ts.add("-2-");
    ts.add("-1-");
    System.out.println(ts);
    System.out.println(ts.contains("-2-"));
    System.out.println(ts.contains("-4-"));

    System.out.println("HashSet");
    Set<String> hs = new HashSet<String>();
    hs.add("-1-");
    hs.add("-3-");
    hs.add("-2-");
    hs.add("-1-");
    System.out.println(hs);
    System.out.println(hs.contains("-2-"));
    System.out.println(hs.contains("-4-"));
  }
}
```

Sortie

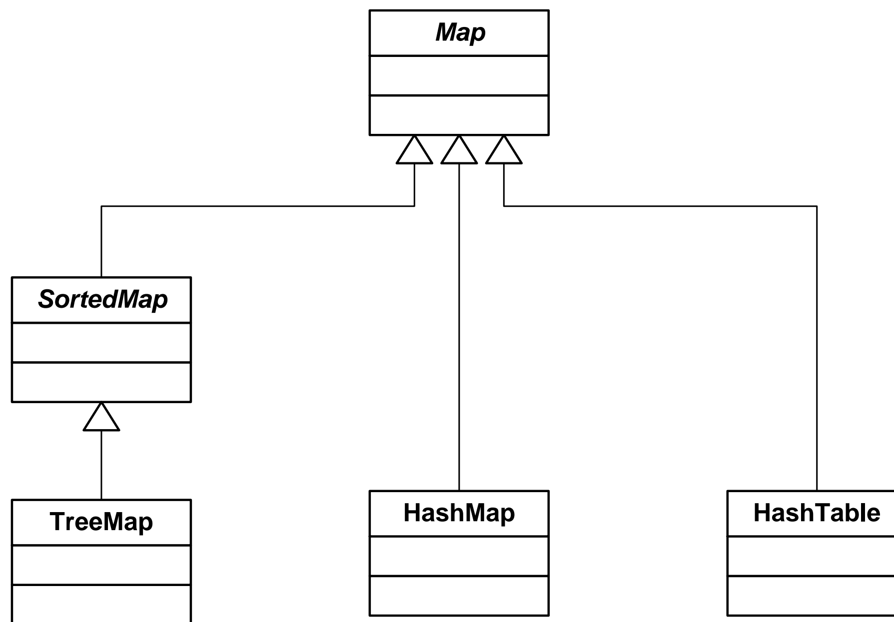
```
TreeSet
[-1-, -2-, -3-]
true
false
HashSet
[-3-, -1-, -2-]
true
false
```

LES COLLECTIONS JAVA ASSOCIATIVES

DEFINITIONS

- Les données sont organisées par paires : clé + donnée.
  `Map<String, String> m = new TreeMap<String, String>();`
  `m.put("Lajoie", "05 46 66 33 50");`
  `"Lajoie"` est la clé, `"05 46 66 33 50"` la donnée.
- La clé ou index sert à retrouver la donnée associée.

HIERARCHIE DES CLASSES DES CONTENEURS ASSOCIATIFS



INTERFACE MAP (T.D.A.)

Principales méthodes de l'interface

Cette interface déclare les méthodes que doit implémenter `SortedMap`, et `HashTable`, ce qui implique que le code d'une application utilisant un ensemble `Set` (le TDA) implémenté avec un `SortedMap` (c'est la S.D.F) est totalement interchangeable avec un `Map` implémenté avec un `HashMap`.

boolean	**containsKey**(Object key)
boolean	**containsValue**(Object value)
boolean	**equals**(Object o)
V	**get**(Object key)
boolean	**isEmpty**()
Set<K>	**keySet**() // ensemble des clefs (K) et le type paramétré
V	**put**(K key, V value) // K et V types parametrés
V	**remove**(Object key)
int	**size**()
Collection<V>	**values**()

CLASSE TREEMAP (S.D.F.)

Un dictionnaire `TreeMap` trie les clés.

- Exemple : annuaire téléphonique utilisant un `TreeMap`

```
Map<String, String> m = new TreeMap<String, String>();
m.put("Lajoie", "05 46 66 33 50");
m.put("Dupond", "06 46 66 33 00");
m.put("Durand", "05 46 66 33 20");
System.out.println(m.keySet());
System.out.println(m.values());

Affiche :
[Dupond, Durand, Lajoie]
[06 46 66 33 00, 05 46 66 33 20, 05 46 66 33 50]

// Annuaire inversé
Map<String, String> m3 = new TreeMap<String, String>();
Set v = m.keySet();
Iterator<String> it = v.iterator();
for (int i = 0; i < m.size(); i++)
{
      String s = it.next();
      m3.put(m.get(s), s);
}
System.out.println(m3.get("05 46 66 33 50"));

Affiche :

Lajoie
```

CLASSE HASHMAP (S.D.F.)

Ne trie pas les clés mais offre un accès 30% plus rapide que le conteneur précédent.
On peut reprendre intégralement les exemples précédents en remplaçant `TreeMap` par `HashMap`

```
Map<String, String> m = new HashMap<String, String>();

m.put("Lajoie", "05 46 66 33 50");
m.put("Dupond", "06 46 66 33 00");
m.put("Durand", "05 46 66 33 20");
System.out.println(m.keySet());
System.out.println(m.values());

Affiche :
[Lajoie, Durand, Dupond]
[05 46 66 33 50, 05 46 66 33 20, 06 46 66 33 00]

// Annuaire inversé
Map<String, String> m3 = new HasMap<String, String>();
Set v = m.keySet();
Iterator<String> it = v.iterator();
for (int i = 0; i < m.size(); i++)
{
      String s = it.next();
      m3.put(m.get(s), s);
}
System.out.println(m3.get("05 46 66 33 50"));

Affiche :

Lajoie
```

L'utilisation de ces classes sera vue en détail dans l'exercice *DesMots*.

LES OPERATIONS ENTREES/SORTIE

Tous les langages de programmation possèdent un mécanisme permettant l'échange de donnée entre la mémoire vive (ram) et la mémoire secondaire (disque). Java ne déroge pas à la règle et propose un mécanisme basé sur les flots. Les flots permettent les échanges entre la mémoire et des entités très différentes comme la mémoire secondaire et la mémoire d'un ordinateur distant. Les échanges se font de façon séquentielle.

LES FLOTS JAVA

Java fait la distinction entre les flots d'entrée et les flots de sortie. Les flots d'entrée permettent la lecture de données depuis une entité externe au programme (les entités sources), réciproquement le flot de sortie permettent l'écriture de données vers une entité externe (les entités destination).
Outre cette distinction les flots sont également caractérisés par la nature des informations échangée. On distingue :

- Les flots d'octets : l'information est codée sur 8 bits, c'est-à-dire avec un type **`byte`**.
- Les flots de caractères : l'information est codée avec un type **`char`** du jeu de caractères UNICODE.

Les classes nécessaires pour manipuler les flots sont contenus dans le paquetage `java.io`.

CONSTRUIRE LES FLOTS JAVA

Les flots construits dérivent tous de quatre classes abstraites :

	Nom du flot d'entrée	*Nom du flot de sortie*
Flots d'octets	`InputStream`	`OutputStream`
Flots de caractères	`Reader`	`Writer`

Les noms des classes qui représentent les entités sources ou destination sont formés en faisant précéder le type de flot par le type de l'entité.

	Entrée	*Sortie*
Fichier de flots d'octets	`FileInputStream`	`FileOutputStream`
	`ByteArrayInputStream`	`ByteArrayOutputStream`
	`PipedInputStream`	`PipedOutputStream`

- java.lang.**Object**
  - java.io.**InputStream** (implements java.io.Closeable)
    - java.io.**ByteArrayInputStream**
    - java.io.**FileInputStream**
    - java.io.**FilterInputStream**
    - java.io.**ObjectInputStream** (implements java.io.ObjectInput, java.io.ObjectStreamConstants)
    - java.io.**PipedInputStream**
    - java.io.**SequenceInputStream**
    - java.io.**StringBufferInputStream**

	Entrée	*Sortie*
Fichier de flots de caractères	`FileReader`	`FileWriter`
	`CharArrayReader`	`CharArrayWriter`
	`StringReader`	`StringWriter`
	`PipedReader`	`PipedWriter`

- java.lang.Object
  - java.io.Reader (implements java.io.Closeable, java.lang.Readable)
    - java.io.BufferedReader
      - java.io.LineNumberReader
    - java.io.CharArrayReader
    - java.io.FilterReader
      - java.io.PushbackReader
    - java.io.InputStreamReader
      - java.io.FileReader
    - java.io.PipedReader
    - java.io.StringReader

Les classes peuvent être composées avec d'autres flots pour apporter de nouvelles fonctionnalités au flot initial. Les règles de formation des noms restent la même.
Le mot `Buffered` permettent des E/S accélérées en accumulant les données dans un tampon (buffer) avant de les envoyer ou de les retourner augmentant ainsi l'efficacité des opérations.

Entrée	*Sortie*
BufferedReader	*BufferedWriter*
LineNumberReader	
PushBackReader	

UTILISER LES FLOTS JAVA

Exemple 1:
On peut définir un flot à partir un objet de type `String`

```
StringReader sr = new StringReader("abcdefg");
```

Qu'on peut lire caractère par caractère avec la méthode :

```
int  read();
```

Ou par blocs de plusieurs caractères à suivre :

```
int  read(char[] cbuf, int off, int len);
```

Application

```
StringReader sr = new StringReader("abcdefg");
char[] buff = new char[5];
sr.read(buff,0, buff.length);
System.out.println(buff);
System.out.println((char)sr.read());
```

Affiche

```
abcde
f
```

Exemple 2:
Définir un flot sur un contenu de page Web puis en lire le contenu (ces classes seront vues par la suite) :

```
URL webURL =
     new URL(
          "http://daniel.tschirhart.free.fr/java/index.htm"
          );
BufferedReader
    line_in = new BufferedReader(
                   new InputStreamReader(webURL.openStream()));

String ligne = line_in.readLine();
```

FLOTS PREDEFINIS

- Java propose des flots prédéfinis dans le package java.io : `in` et `out`
  - le flot standard in relie le clavier à la mémoire vive (programme). Exemple :
    `char c = System.in.read();`
  - le flot standard out relie la mémoire vive à l'écran. Exemple
    `System.out.println("Exemple");`

LES FICHIERS

- Les fichiers utilisent les flots et sont caractérisés par la nature des informations échangées. On distingue :
  - Les fichiers d'octets (mot de 8 bits), c'est-à-dire de *bytes*,
  - Les fichiers de caractères UNICODE,
  - Les fichiers composés de lignes de caractères terminés par le caractère CR,
  - Les fichiers composés d'éléments de type primitif,
  - Les fichiers d'objets.

INSTANCIER DES OBJETS FICHIERS

Les classes qui permettent de manipuler les fichiers possèdent des constructeurs auxquels on spécifie :

- le nom du fichier à ouvrir,
  - le nom du fichier est fourni au constructeur sous la forme d'un objet *String, File* ou *FileDescriptor.*
- le sens d'échange de donnée (lecture ou écriture),
  - Une ouverture en mode lecture, positionne l'index de lecture en début de fichier. Le fichier doit exister et être accessible.
    Toute erreur déclenche une exception *`FileNotFoundException`* ou *`SecurityException`*.
  - Une ouverture en mode écriture sur un fichier existant vide ce dernier sauf si on précise que l'ouverture de fait en mode ajout. Si le fichier n'existe pas, il est crée (si les droits dans le répertoire l'autorise). Dans le cas contraire une exception *`SecurityException`* est émise.
- le mode d'ouverture (création, ajout, …).

CHEMIN DES FICHIERS.

Avec UNIX le chemin d'un fichier est spécifié par un objet *String* ou les différents niveaux de répertoires sont séparés par le caractère '/'
Avec Windows le chemin est spécifié est spécifié par un objet *String* ou les différents niveaux de répertoires sont séparés par le caractère '\'. Comme ce caractère est un méta caractère pour les String, il faut le doubler.
Exemples

```
// UNIX : ouverture d'un fichier d'octets
FileInputStream f = new FileInputStream("java/exemple.class");

// Windows : ouverture d'un fichier de caractères
FileReader fr  = new FileReader("c:\\java\\exemple.java");

// Windows : ouverture d'un fichier de caractères chemin UNC
FileReader fr2 = new FileReader("\\\\machine\\java\\exemple.java");

// Ouverture en mode ajout
FileOutputStream fos = new FileOutputStream FileReader("fichier1", true);
```

FICHIERS D'OCTETS

Les fichiers d'octets sont des instances des classes `FileInputStream` et `FileOutputStream`.
Les opérations élémentaires sont la lecture et l'écriture séquentielle :

Entrée	*Sortie*
`int read()`	*`void write()`*
`int read(byte[] b)`	*`void write(byte[] b)`*

Lorsque la fin du fichier est atteinte, *read* retourne -1.

Exemple :

```
FileInputStream is = new FileInputStream(".\\src\\Flots.java");
ou
FileInputStream is = new FileInputStream("src\\Flots.java"); // même chemin
FileOutputStream os = new FileOutputStream("Flots.txt");
byte b;
while ((b = (byte) is.read()) != -1)
  os.write(b);
is.close();
os.close();
```

FICHIERS DE CARACTERES

Les fichiers de caractères sont des instances des classes `FileReader` et `Filewriter`.
Les opérations élémentaires sont la lecture et l'écriture séquentielle héritée des classes `FileInputStream` et `FileOutputStream`

```
java.lang.Object
  └ java.io.Reader
      └ java.io.InputStreamReader
          └ java.io.FileReader
```

```
FileReader is = new FileReader("\\src\\Flots.java");
FileWriter os = new FileWriter("Flots.txt");
int val;
while ((val = (int) is.read()) != -1)
  os.write(b);
is.close();
os.close();
```

FICHIERS DE CARACTERES LECTURE/ECRITURE LIGNES PAR LIGNES

Similaire à la lecture caractères par caractères, sauf que celle-ci s'effectue ligne par ligne.
Ne pas oublier de rajouter explicitement '\n' pour marquer la fin de ligne lors de l'écriture.

```
import java.io.*;
public class Flots5
{
  public static void main(String[] args)
  {
    try
    {
      BufferedReader line_in =  new BufferedReader(
                                new FileReader(
                                new File(".\\src\\Flots5.java")));
      BufferedWriter line_out = new BufferedWriter(
                                new FileWriter(
                                new File("Flots5.java")));

      String ligne;
      while ((ligne = line_in.readLine()) != null)
        line_out.write(ligne + "\n");
      line_out.close();
      line_in.close();
    }
    catch (IOException e)
    {
      e.printStackTrace();
    }
  }
}
```

FICHIERS D'ELEMENTS PRIMITIFS

Les fichiers d'éléments primitifs sont construit par composition d'objets `FileInputStream` ou `FileOutputStream` avec des objets `DataInputStream` ou `DataOutputStream`.

```
DataOutputStream os = new DataOutputStream(new FileOutputStream("data.bin"));
```

Les principales méthodes disponibles sont données ci-dessous.

int	**read**(byte[] b) Reads some number of bytes from the contained input stream and stores them into the buffer array `b`.
int	**read**(byte[] b, int off, int len) Reads up to `len` bytes of data from the contained input stream into an array of bytes.
boolean	**readBoolean**() See the general contract of the `readBoolean` method of `DataInput`.
byte	**readByte**() See the general contract of the `readByte` method of `DataInput`.
char	**readChar**() See the general contract of the `readChar` method of `DataInput`.
double	**readDouble**() See the general contract of the `readDouble` method of `DataInput`.
int	**readInt**() See the general contract of the `readInt` method of `DataInput`.
long	**readLong**() See the general contract of the `readLong` method of `DataInput`.
short	**readShort**() See the general contract of the `readShort` method of `DataInput`.
int	**skipBytes**(int n) See the general contract of the `skipBytes` method of `DataInput`.

La fin du fichier est détectée par l'exception `EOFException`.
Exemple :

```
// Ecrit un tableau de 4 entiers puis le relit
int[] t = { 1, 2, 3, 4 };
// Ecrit le tableau
DataOutputStream os = new DataOutputStream(new FileOutputStream("data.bin"));
for (int j=0; j < t.length; j++)
  os.writeInt(t[j]);
os.close();

// Le code ci-dessous lit le fichier
DataInputStream is = new DataInputStream(new FileInputStream("data.bin"));
int b;

try
{
   for (;;) // La fin du fichier déclenche l'exception EOFException
   {
     b = is.readInt();
     System.out.println(b);
   }

}
catch (EOFException e)
{
  System.out.println("Fin de fichier");
  is.close();
}
catch (Exception e)
{
  System.out.println("Autre exception");
}
}
```

Fichiers d'objets

La mémorisation d'un objet permet d'enregistrer son état pour le reprendre plus tard. Cette opération portant également le nom de **sérialisation** est une sorte de photographie d'un objet à un instant donné.
Par exemple il est possible d'enregistrer en une seule opération un objet de type `TreeMap`.

L'opération de dé-sérialisation permet de lire directement des objets depuis son image enregistrée.
Pour rendre un objet *sérialisable* il faut dériver l'interface *Serializable* (il n'y a aucune méthode à implémenter).
Pour écrire un objet il faut créer un objet *ObjectOutputStream* par composition avec un *ObjectOutputStream.*
ObjectOutputStream ofR = new ObjectOutputStream(new FileInputStream("Promo.dat"));

Pour lire un objet sérialisé il faut créer un objet *ObjectInputStream* par composition avec un *FileInputStream.*
ObjectInputStream ofR = new ObjectInputStream(new FileInputStream("Promo.dat"));

Exemple :

```
import java.io.*;

class Etudiant implements Serializable
{
   String Nom;
   int Promo;
   Etudiant(String n, int p)
   {
     Nom = n; Promo = p;
   }
   public String toString()
   {
      return Nom + ": " + Promo;
   }
}

public class TestSerialisable
{
   public static Vector Lire() throws IOException, ClassNotFoundException
  {
      ObjectInputStream ofR = new ObjectInputStream(new FileInputStream("Promo.dat"));
      return  (Vector)ofR.readObject();
  }

     public static void Ecrire(Vector vector) throws IOException
  {
    ObjectOutputStream ofW = new ObjectOutputStream(new FileOutputStream("Promo.dat"));
    ofW.writeObject(vector);
  }

  public static void main(String[] args)
  {
    Vector<Etudiant> v = new Vector<Etudiant>(); // NOTA: Vecteur est marqué Serializable
    v.add(new Etudiant("Dupond", 2003));
    v.add(new Etudiant("Lajoie", 2002));
    v.add(new Etudiant("Durand", 2003));
    v.add(new Etudiant("Luc", 2005));
    try
    {
      Ecrire(v);                    // Ecrire le vecteur d'étudiants
      System.out.println(Lire());  // Lire le vecteur d'étudiants
    }
    catch (ClassNotFoundException c) { System.out.println("Exceptions :" +  c); }
    catch (IOException e)            { System.out.println("Exceptions :" +  e); }
    catch (Exception e)              { System.out.println("Exceptions :" +  e); }
}
```

LA PROGRAMMATION GRAPHIQUE

API GRAPHIQUES DE JAVA

Java AWT (Abstract Windows Toolkit) :

- Utilise l'API graphique sous-jacente de la plateforme d'accueil.
- N'utilise que les éléments communs à toutes les plateformes.
- Une application AWT possède le *look* de la plateforme d'accueil.

Swing

- Java Swing est écrit entièrement en Java.
- Plus lourd, lent et plus gourmand en mémoire.
- Possède de nombreux contrôles.
- look indépendant de la plateforme d'accueil.

CLASSES PRINCIPALES DE L'API AWT

- `Canvas` permet de dessiner dans un cadre de fenêtre
- `Frame` permet d'afficher une fenêtre
- `Dialog` permet d'afficher une boite de dialogue
- `Graphics` qui fourni l'accès aux outils de dessin.
- `Button` permet de créer des boutons,
- `Checkbox` permet de créer des cases à cocher,
- `Label` permet d'insérer du texte sur une fenêtre,
- `TextField` permet de saisir du texte,
- `List` permet de créer des boites à liste.

CREATION D'UNE FENETRE

Deux techniques possibles :

- Surclasser (dériver) la classe `Frame`. Dans ce cas on programme l'application dans une fonction membre de la et on instancie la classe dans `main()`. Solution recommandée pour sa simplicité, mais ne convient pas aux applications qui nécessitent d'hériter d'autres classes (Java ne supporte pas l'héritage multiple).

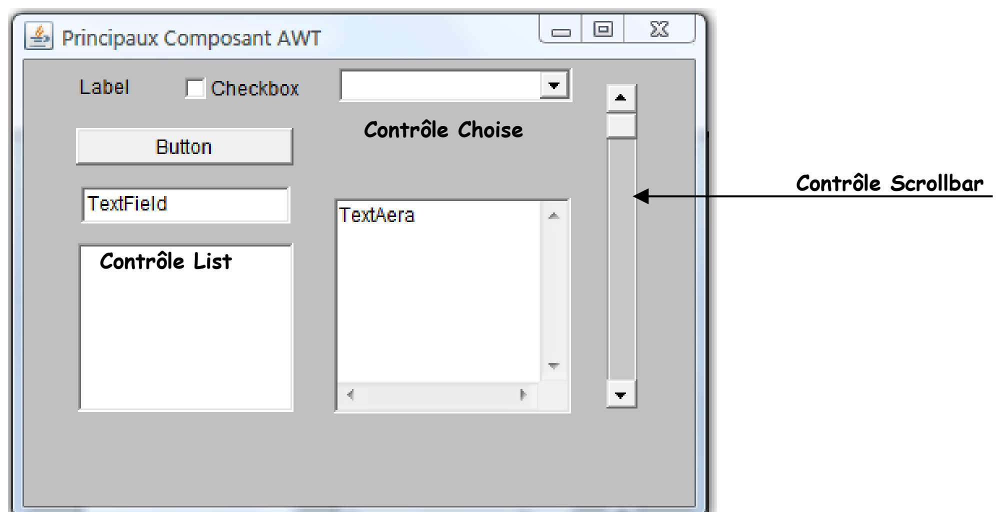
```
import java.awt.*;
class Calculette extends Frame
{
  private final static int LG = 200;
  private final static int HT = 180;

  public Calculette()
  {
    setTitle("Calculette Euro");
    setSize(LG, HT);
    ...
  }
   public static void main(String[] arg)
   {
     new Calculette().setVisible();
   }
}
```

- Instancier la classe frame directement dans `main()`.

```
import java.awt.*;
public class Figures
{
	private final static int HT = 300;
	private final static int LG = 300;
	...
	public static void main(String[] arg)
	{
		Frame F = new Frame();
		F.setTitle("Figures"); // met le titre
		F.setSize(HT, LG);     // taille de la fenêtre
		F.show();              // affiche la fenêtre
	}
}
```

PRINCIPAUX ELEMENTS D'IHM

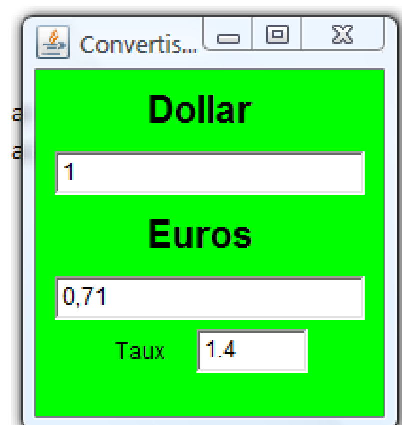
Les objets d'interface graphique s'appellent des contrôles. Ils génèrent des événements aux quels on abonne des méthodes appelées gestionnaires d'événements ou écouteurs.



LES LABELS

Les labels sont des zones de texte pouvant être lus ou écrits par programmes.
Les labels génèrent un seul événement (label activé).

Mise en place d'un label




```
// Titre de la zone de saisie
Label labelDollar = new Label("Dollar");
...
labelDollar.setFont(new Font("", Font.Bold, 20);
add(labelDollar);
```

Principaux constructeurs

Constructor Summary
Label() Constructs an empty label.
Label(String text) Constructs a new label with the specified string of text, left justified.
Label(String text, int alignment) Constructs a new label that presents the specified string of text with the specified alignment.

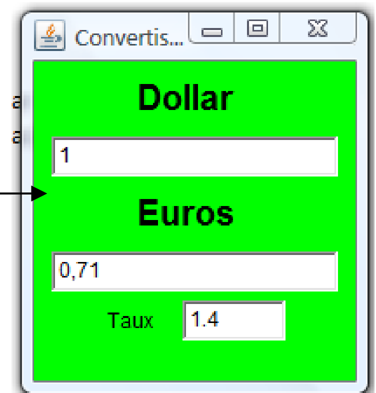
Method Summary	
String	**getText**() Gets the text of this label.
void	**setText**(String text) Sets the text for this label to the specified text.

LES ZONES D'EDITION

Les zones d'édition sont des zones permettant la saisie de données au clavier. Ces zones constituent un mini éditeur de texte (gestion du curseur, du presse papier, ...).

Mise en place d'une zone d'édition

```
TextField saisieDollar;
...
saisieDollar = new TextField(20);
add(saisieDollar);
```



Principaux constructeurs

Constructor Summary
TextField() Constructs a new text field.
TextField(int columns) Constructs a new empty text field with the specified number of columns.
TextField(String text) Constructs a new text field initialized with the specified text.
TextField(String text, int columns) Constructs a new text field initialized with the specified text to be displayed, and wide enough to hold the specified number of columns.

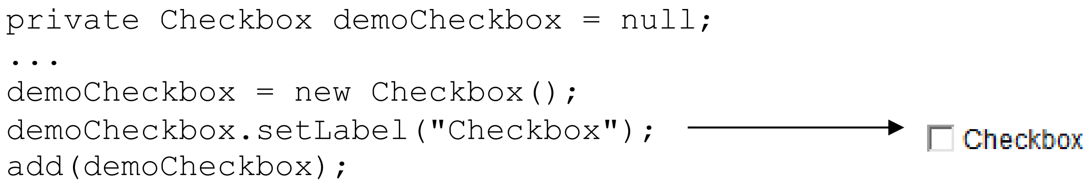
Principales méthodes

Method Summary	
void	**addActionListener**(ActionListener l) Adds the specified action listener to receive action events from this text field.
void	**setColumns**(int columns) Sets the number of columns in this text field.
void	**setEchoChar**(char c) Sets the echo character for this text field.
void	**setText**(**String** t) Sets the text that is presented by this text component to be the specified text.
String	**getText**() Sets the text that is presented by this text component to be the specified text.

LES CASES A COCHER

Les cases à cocher permettent de choisir de valider ou non des éléments quelconques.

Mise en place d'une case à cocher



```
private Checkbox demoCheckbox = null;
...
demoCheckbox = new Checkbox();
demoCheckbox.setLabel("Checkbox");
add(demoCheckbox);
```


Principaux constructeurs

Constructor Summary
Checkbox()   Creates a check box with an empty string for its label.
Checkbox(String label)   Creates a check box with the specified label.
Checkbox(String label, boolean state)   Creates a check box with the specified label and sets the specified state.
Checkbox(String label, boolean state, CheckboxGroup group)   Constructs a Checkbox with the specified label, set to the specified state, and in the specified check box group.
Checkbox(String label, CheckboxGroup group, boolean state)   Creates a check box with the specified label, in the specified check box group, and set to the specified state.

Principales méthodes

Method Summary	
void	**addItemListener**(ItemListener l) Adds the specified item listener to receive item events from this check box.
String	**getLabel**() Gets the label of this check box.
boolean	**getState**() Determines whether this check box is in the "on" or "off" state.
void	**setCheckboxGroup**(CheckboxGroup g) Sets this check box's group to the specified check box group.
void	**setLabel**(String label) Sets this check box's label to be the string argument.
void	**setState**(boolean state) Sets the state of this check box to the specified state.

Exemple

```
if (demoCheckbox.getState() == true)
     // La case à cochée est cochée
```

LES BOUTONS

Les boutons sont des éléments d'IHM utilisés généralement pour valider des actions.

Mise en place du bouton

```
private Button demoButton = null;
...
demoButton = new Button();
demoButton.setLabel("Bouton");
demoButton .setSize(new Dimension(175,35));
add(demoButton);
```

Constructeurs et principales méthodes

Constructor Summary
Button()   Constructs a button with an empty string for its label.
Button(String label)   Constructs a button with the specified label.

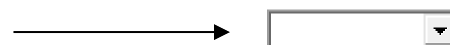
Method Summary	
void	**addActionListener**(ActionListener l) Adds the specified action listener to receive action events from this button.
String	**getActionCommand**() Returns the command name of the action event fired by this button.
String	**getLabel**() Gets the label of this button.
void	**setActionCommand**(String command) Sets the command name for the action event fired by this button.
void	**setLabel**(String label) Sets the button's label to be the specified string.

LES COMBOS

Les boites combinées permettent d'effectuer un choix parmi n éléments.

Mise en place d'un combo

```
private Choise demoChoise = null;

demoChoise = new Choise();
demoChoise.setSize(100,100);
add(demoChoise);
```



Constructeur et principales méthodes

Constructor Summary
Choice()   Creates a new choice menu.

Method Summary	
void	**add**(String item) Adds an item to this Choice menu.
void	**addItemListener**(ItemListener l) Adds the specified item listener to receive item events from this Choice menu.
void	**addNotify**() Creates the Choice's peer.
String	**getItem**(int index) Gets the string at the specified index in this Choice menu.
int	**getItemCount**() Returns the number of items in this Choice menu.

int	getSelectedIndex() Returns the index of the currently selected item.
String	getSelectedItem() Gets a representation of the current choice as a string.
void	insert(String item, int index) Inserts the item into this choice at the specified position.
void	remove(int position) Removes an item from the choice menu at the specified position.
void	remove(String item) Removes the first occurrence of item from the Choice menu.
void	removeAll() Removes all items from the choice menu.
void	select(int pos) Sets the selected item in this Choice menu to be the item at the specified position.
void	select(String str) Sets the selected item in this Choice menu to be the item whose name is equal to the specified string.

LES BOITES A LISTES

Les boites à listes permettent de mémoriser des chaines de caractères. Cet élément est par exemple utilisé dans la calculette complexe.

Mise en place d'une boite à listes

```
List liste = null;
...
liste = new List();
liste.add("Liste ligne 1");
liste.add("Liste ligne 2");
liste.add("Liste ligne 3");
add(liste);
```

Liste ligne 1
Liste ligne 2
Liste ligne 3

Constructeurs et principales méthodes

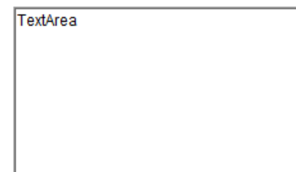
Constructor Summary
List() Creates a new scrolling list.
List(int rows) Creates a new scrolling list initialized with the specified number of visible lines.
List(int rows, boolean multipleMode) Creates a new scrolling list initialized to display the specified number of rows.

Method Summary	
void	add(String item) Adds the specified item to the end of scrolling list.
void	add(String item, int index) Adds the specified item to the the scrolling list at the position indicated by the index.
void	addActionListener(ActionListener l) Adds the specified action listener to receive action events from this list.
void	addItemListener(ItemListener l) Adds the specified item listener to receive item events from this list.
void	deselect(int index) Deselects the item at the specified index.
String	getItem(int index) Gets the item associated with the specified index.
int	getItemCount() Gets the number of items in the list.
String[]	getItems() Gets the items in the list.

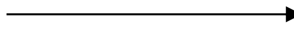
String	**getSelectedItem**() Gets the selected item on this scrolling list.
String[]	**getSelectedItems**() Gets the selected items on this scrolling list.
int	**getVisibleIndex**() Gets the index of the item that was last made visible by the method `makeVisible`.
boolean	**isIndexSelected**(int index) Determines if the specified item in this scrolling list is selected.
void	**makeVisible**(int index) Makes the item at the specified index visible.
void	**remove**(int position) Removes the item at the specified position from this scrolling list.
void	**remove**(String item) Removes the first occurrence of an item from the list.
void	**removeAll**() Removes all items from this list.
void	**replaceItem**(String newValue, int index) Replaces the item at the specified index in the scrolling list with the new string.
void	**select**(int index) Selects the item at the specified index in the scrolling list.

LES TEXTAREA

Un `TextArea` une zone d'édition multi lignes.



Mise en place d'un texteAera



```
TextAera demoTextAera = null;
...
demoTextAera  = new TextAera("TextAera", 4, 4, TextAera.SCROLLBARS_NONE);
add(demoTextAera);
```

Constructeurs

Constructor Summary
TextArea() Constructs a new text area with the empty string as text.
TextArea(int rows, int columns) Constructs a new text area with the specified number of rows and columns and the empty string as text.
TextArea(String text) Constructs a new text area with the specified text.
TextArea(String text, int rows, int columns) Constructs a new text area with the specified text, and with the specified number of rows and columns.
TextArea(String text, int rows, int columns, int scrollbars) Constructs a new text area with the specified text, and with the rows, columns, and scroll bar visibility as specified.

Principales méthodes

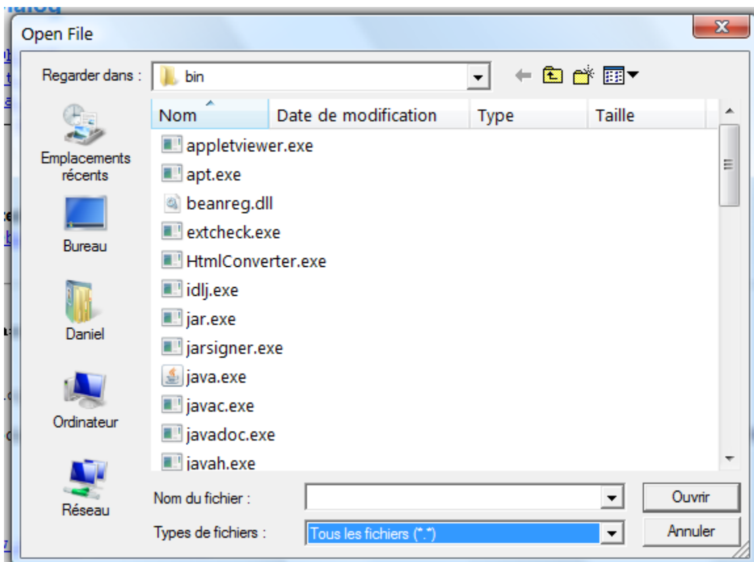
Method Summary	
void	append(String str) Appends the given text to the text area's current text.
int	getColumns() Returns the number of columns in this text area.
int	getRows() Returns the number of rows in the text area.
String	getText()
int	getScrollbarVisibility() Returns an enumerated value that indicates which scroll bars the text area uses.
void	insert(String str, int pos) Inserts the specified text at the specified position in this text area.
void	replaceRange(String str, int start, int end) Replaces text between the indicated start and end positions with the specified replacement text.
void	setColumns(int columns) Sets the number of columns for this text area.
void	setRows(int rows) Sets the number of rows for this text area.

FILEDIALOG

FileDialog est un contrôle permettant l'accès aux fichiers et répertoires.

Mise en place d'un filedialog

```
FileDialog openDialog = new FileDialog(this, "Open File", FileDialog.LOAD);
openButton.addActionListerner(this); // mise en place d'un écouteur
```



Principaux constructeurs et méthodes

Constructor Summary
TextArea() Constructs a new text area with the empty string as text.
TextArea(int rows, int columns) Constructs a new text area with the specified number of rows and columns and the empty string as text.
TextArea(String text) Constructs a new text area with the specified text.
TextArea(String text, int rows, int columns) Constructs a new text area with the specified text, and with the specified number of rows and columns.
TextArea(String text, int rows, int columns, int scrollbars) Constructs a new text area with the specified text, and with the rows, columns, and scroll bar visibility as specified.

Method Summary	
void	append(String str) Appends the given text to the text area's current text.
int	getColumns() Returns the number of columns in this text area.
int	getRows() Returns the number of rows in the text area.
String	getText()
int	getScrollbarVisibility() Returns an enumerated value that indicates which scroll bars the text area uses.
void	insert(String str, int pos) Inserts the specified text at the specified position in this text area.
void	replaceRange(String str, int start, int end) Replaces text between the indicated start and end positions with the specified replacement text.
void	setColumns(int columns) Sets the number of columns for this text area.
void	setRows(int rows) Sets the number of rows for this text area.

GESTIONNAIRE DE L'EVENEMENT ASSOCIE

```
public void actionPerformed(ActionEvent ae)
{
  if (ae.getSource()==openButton)
  {
    openDialog.setVisible(true);
    String dir = openDialog.getDirectory();
    System.out.println(dir);
  }
}
```

DISPOSER LES CONTROLES DANS LA FENETRE

Java propose plusieurs façons de disposer un contrôle dans la fenêtre. Les principales sont :

- Disposition libre,
- Disposition dans une grille,
- Disposition suivant des coordonnées géographiques (Sud, Nord, Est, Ouest),
- Disposition dans l'ordre de déclaration (la méthode la + simple).

EXEMPLE

```
setLayout(new FlowLayout()); // disposition dans l'ordre de déclaration
setLayout(null);             // disposition libre
```

METTRE EN PLACE LES CONTROLES DANS LA FENETRE

- **Déclarer les contrôles comme attribut de façon à ce qu'ils puissent être accessibles depuis plusieurs méthodes**.
- Instancier le contrôle.
- Associer (si nécessaire) une police de caractère au contrôle.
- Enregistrer le contrôle dans la fenêtre

EXEMPLE

```
import java.awt.*;

public class Calculette extends Frame
{
  private TextField saisieFrancs;
  ...
  public Calculette()
  {
    ...
    saisieFrancs = new TextField(20);
    labelEuros.setFont(new Font("", Font.BOLD, 20));
    add(saisieFrancs);
    ...
  }
}
```

CARACTERISTIQUES COMMUNES AUX CONTROLES

COULEUR D'AVANT PLAN ET ARRIERE PLAN

```
setForeground
setBackground
```

Exemples :

```
label1.setForeground(Color.RED);
this.setBackground(new Color(0xE0F8FE)); // couleur RVB
bouton.setBackground();
bouton.setBackground(Color.PINK);
bouton.setForeground(Color.BLUE);
```

TAILLE DU CONTROLE (LARGEUR, HAUTEUR)

```
setSize
```

Exemples :

```
this.setSize(763, 373);
bouton.setSize(100, 30);
```

NOM INTERNE

Ces noms ne sont pas affichés, ils peuvent être utilisés pour identifier un contrôle de façon dynamique.

```
setName, getName,
```

Exemples :

```
this.setName("Ma Fenêtre");
bouton.setName("Bouton 1");
```

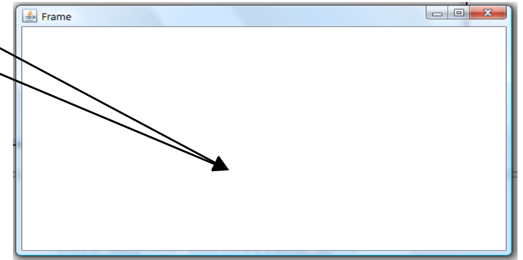
POSITION ET DIMENSION

Permet de définir la position et la taille du contrôle dans la fenêtre

```
setBounds
```

Exemple :

```
bouton.setBounds(new Rectangle(x,   y, largeur, hauteur));
```



POLICE

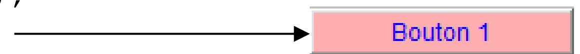
Pour des raisons de portabilité on ne peut pas utiliser les polices définies dans le S.E hôte. Le JDK défini des polices génériques qu'il substitue aux polices les plus approchantes du S.E. hôte. Ainsi le JDK ne connait que les polices génériques suivantes :

- Dialog
- DialogInput
- Monospaced
- Serif
- SansSerif

La police à utiliser sur un contrôle se défini à l'aide de la fonction ***setFont***

Exemples :

```
bouton.setFont(new Font("",Font.BOLD,20));
bouton.setFont(new Font("Monospaced",
                        Font.PLAIN |
                        Font.BOLD |
                        Font.ITALIC,20));
```



INTERAGIR AVEC L'UTILISATEUR : LA PROGRAMMATION EVENEMENTIELLE

- Avec la programmation séquentielle, c'est le système d'exploitation qui a l'initiative du dialogue avec l'utilisateur. Le programme doit être prêt à répondre aux sollicitations du système d'exploitation lui-même sollicité par l'utilisateur.
- L'application dispose de fonctions spéciales nommées écouteurs. Ces fonctions sont à l'écoute d'un type d'événements particuliers à chaque écouteur..

PRINCIPE DE LA PROGRAMMATION EVENEMENTIELLE

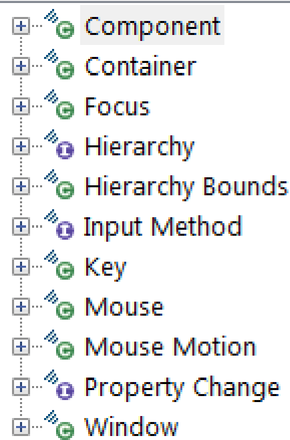
1. Toute action de la part de l'utilisateur (événement) est détectée par le système d'exploitation puis est transmis à la machine Java.
2. Le programme Java a abonné au préalable une fonction à l'événement considéré. La fonction abonnée s'appelle un gestionnaire d'événement ou écouteur.
3. Lorsque surviennent des événements, le système d'exploitation appelle la machine Java. Celle les filtres et appelle la fonction abonnée à cet événement.

GESTIONNAIRES D'EVENEMENTS OU ECOUTEUR

- Pour qu'un événement remonte vers l'application, l'objet contrôlé par cet événement doit s'abonner à cet événement.

TYPE D'EVENEMENTS

Java propose des abonnements à une grande variété d'événements (non exhaustif)



ABONNEMENT AUX EVENEMENTS

- Pour que le gestionnaire d'événement ou écouteur soit appelé, il faut abonner le contrôle au gestionnaire correspondant. Nota on peut abonner un écouteur à plusieurs événements.
- Le nom des gestionnaires d'événement ou écouteur est spécifié :
  - soit dans une interface, auquel cas il faut dériver la classe concernée de l'interface et définir toutes les fonctions de cette interface (même si elles ne font rien)
  - l'écouteur peut être présent dans une classe proposant un comportement par défaut. Dans ce cas on surclasse cette dernière et on redéfinit la méthode désirée.
- Chaque catégorie d'événement possède une méthode permettant d'abonner une fenêtre à l'événement du contrôle placé dans cette dernière.

SYNTAXE DES ABONNEMENTS

- < réf source événement>.<méthode>(<ref obj à abonner>)
  - < réf source événement> : référence de l'objet générateur d'événements
  - < méthode> : **add**<nom événement>**Listener** Exemple : add**Key**Listener
  - <ref obj à abonner> : référence de l'objet abonné.
    - Si les gestionnaires d'événements appartiennent à la classe *abonnante* la référence de l'objet à abonner est this
    - Si les gestionnaires d'événements appartiennent une classe différente de la classe *abonnante* la référence de l'objet est la référence d'un objet de cette classe.

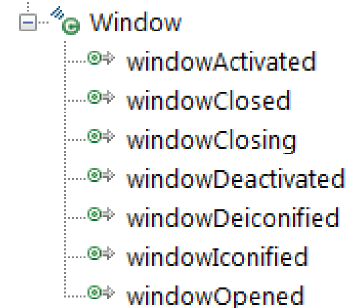
QUELS EVENEMENTS POUR QUELS CONTROLES ?

- Le choix d'un événement associé à un contrôle est une question de logique.
- La lecture de la documentation associée à la classe fournit dans le cas général le nom du gestionnaire le plus adapté.
- Plusieurs gestionnaires utilisables sont définis dans les classes parentes (naviguer dans la documentation de façon intelligente), mais le choix reste toujours une question de bon sens !
- Eclipse fournit une aide précieuse en présentant tous les gestionnaires présent dans la classe et dans les classes parentes, mais le choix reste toujours une question de bon sens.

EVENEMENTS LIES A LA FENETRE

Les événements liés à la fenêtre sont particuliers dans la mesure où les écouteurs sont prédéfinit.

- Un événement peut se déclencher par :
  - L'activation de la fenêtre
  - La fermeture effective de la fenêtre
  - La fermeture de la fenêtre
  - La désactivation de la fenêtre
  - Le passage de l'état d'icône à l'état ouvert
  - Le passage à l'état d'icône
  - L'ouverture de la fenêtre



Abonnement

- Abonnement par **addWindowListener**

Gestionnaires d'événements (ou écouteurs)

Comment définir les écouteurs

- Ils peuvent être défini en implémentant l'interface **WindowListener**,
- ou en dérivant la classe **WindowAdapter** au quel cas on surchage la méthode appropriée (voir exemples plus loin).

Signature des écouteurs disponibles

```
public void windowClosing(WindoxEvent e)
public void windowActivated(WindoxEvent e)
public void windowClosed(WindoxEvent e)
public void windowDeactivated(WindoxEvent e)
public void windowDeiconified(WindoxEvent e)
public void windowOpened(WindoxEvent e)
```

Exemple mise en place d'un gestionnaire d'événements liés à la fenêtre et définition de l'écouteur

Méthode 1 : implémenter l'interface **WindowListener**

- S'abonner à l'événement WindowListener

```
this.addWindowListener(this);
```

- Définir les toutes méthodes de l'interface WindowListener

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

public class Exemple1 extends Frame implements WindowListener
{
  public Exemple1()
  {
    this.setTitle("E1");
    this.setLayout(new FlowLayout());
    this.setSize(400, 200);
    this.addWindowListener(this);
    this.setVisible(true);
  }
  public void windowClosing(WindowEvent e)     { System.exit(0); }
  public void windowActivated(WindowEvent e)   { }
  public void windowClosed(WindowEvent e)      { }
  public void windowDeactivated(WindowEvent e) { }
  public void windowDeiconified(WindowEvent e) { }
  public void windowIconified(WindowEvent e)   { }
  public void windowOpened(WindowEvent e)      { }
}
```

Méthode 2 : dériver la classe **WindowAdapter**

Les concepteurs de la bibliothèque AWT ont prévu un mécanisme alternatif très astucieux consistant à faire hériter une classe abstraite (dans le cas présent `WindowAdapter`) de l'interface `WindowListener`. Cette classe définit toutes les fonctions de l'interface par des fonctions vides (comme la plupart des gestionnaires définis dans l'exemple précédent). La classe `WindowAdapter` est rendue abstraite (par le préfix `abstract`) pour interdire toute instanciation directe de la classe, instanciation ne présentant par ailleur aucun intérêt (la classe ne comporte que des fonctions vides). Par contre en héritant de cette classe vous n'aurez besoin que de redéfinir la ou les fonctions nécessaires pour l'application concernée.

Voici le code illustrant ce mécanisme :

```
interface WindowListener
{
  void windowActivated(WindowEvent e);
  void windowClosed    (WindowEvent e);
  void windowClosing   (WindowEvent e);
  // ...
}
abstract class WindowAdapter implements WindowListener
{
  public void windowActivated(WindowEvent e) { /* vide */ }
  public void windowClosed    (WindowEvent e) { /* vide */ }
  public void windowClosing   (WindowEvent e) { /* vide */ }
  // ...
}
```

Pour mettre en place ce mécanisme vous devez :

- Vous abonner à l'événement `WindowListener` (ligne 10) en passant à la méthode l'objet dans le quel se trouve le gestionnaire d'événement.
- Dériver la classe `WindowAdapter` dans une classe qui comportera l'écouteur (ligne 17) et surcharger les ou la méthode(s) dont vous voulez modifier le comportement par défaut (ligne 19).

```
1.      import java.awt.*;
2.      import java.awt.event.*;
3.
4.      public class Exemple2 extends Frame
5.      {
6.        public Exemple2()
7.        {
8.          this.setTitle("E2");
9.          this.setLayout(new FlowLayout());
10.         this.setSize(400, 200);
11.         this.addWindowListener(new GestionFenetre());
12.       }
13.       public static void main(String[] args)
14.       {
15.          new Exemple2().setVisible(true);
16.       }
17.     }
18.
19.     class GestionFenetre extends WindowAdapter
20.     {
21.       public void windowClosing(WindowEvent e) { System.exit(0); }
22.     }
```

Méthode 3 : définir un écouteur dans une classe anonyme

Le langage Java permet de définir des classes ne portant pas de nom : les classes anonymes.
Nota : cette technique est utilisée dans l'éditeur d'IHM Visual Editor.

<u>Comment définir une classe anonyme ?</u>

La définition de ce type de classe est similaire à celle d'une classe « normale ».
Exemple : voici une classe X « normale » dérivant de la classe Y et définissant une fonction qui ne fait rien :

```
class X extends Y
{
  public void uneFonctionQuiNeFaitRien() { }
}
```

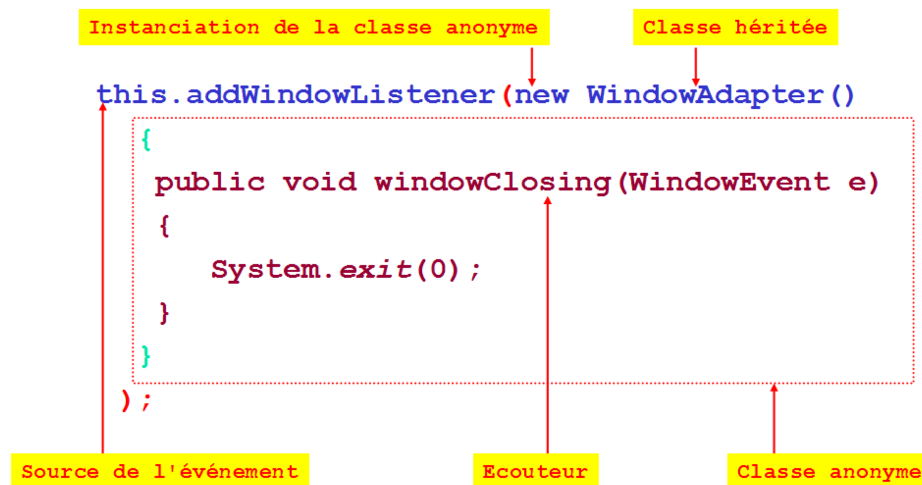
Voici une classe anonyme dérivant de la classe Y et définissant une fonction qui ne fait rien :

```
extends Y
{
  public void uneFonctionQuiNeFaitRien() { }
}
```

Instancier cette classe anonyme s'écrit :

```
new Y()
{
       public void uneFonctionQuiNeFaitRien() { }
}
```

Nota : l'instanciation doit définir en même temps la fonction membre de la classe.
Ces classes héritent de la classe parent et sont utilisées pour redéfinir une des méthodes de la classe parent.
L'intérêt de cette technique est de définir l'écouteur (l'abonné) directement après l'abonnement à un événement. Voici la syntaxe de l'abonnement :

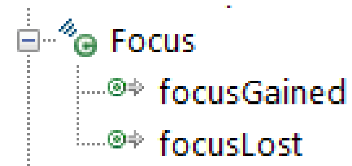


L'exemple ci-dessus redéfinit la méthode windowClosing de la classe WindowAdapter (fonction vide dans cette classe).

```
import java.awt.*;
import java.awt.event.*;
public class Exemple3 extends Frame
{
  public Exemple3()
  {
    this.setTitle("E3");
    this.setLayout(new FlowLayout());
    this.setSize(400, 200);
    this.addWindowListener(new WindowAdapter()
    {
      public void windowClosing(WindowEvent e) { System.exit(0); }
    });
  }
  public static void main(String[] args) { new Exemple3().setVisible(true);  }
}
```

EVENEMENTS ET ABONNEMENTS

Evénements déclenchés par l'activation du contrôle (Focus)



- le contrôle devient actif `focusGained`
- le contrôle devient inactif `focusLost`

Abonnement

- `par addFocusListener`

Ecouteurs

```
public void focusLost  (FocusEvent e)
public void focusGained(FocusEvent e)
```

Définition des écouteurs

- en dérivant l'interface `FocusListener`
- en surclassant la classe `FocusAdapter`
- dans une classe anoyme.

Exemple 1 : dériver l'interface FOCUSLISTENER

```
public class Calculette    extends Frame
                           implements KeyListener, FocusListener, WindowListener
{
  // ...
  TextField saisieDollar, saisieEuros;

  Calculette()
  {
    // ...
    saisieDollar.addFocusListener(this);
    // ...
  }
  // Appelé lorsque l'objet zone d'édition reçoit le focus
  public void focusGained(FocusEvent arg0)
  {
    saisieDollar.setText("");
    saisieEuros.setText("");
  }
  // ...
}
```

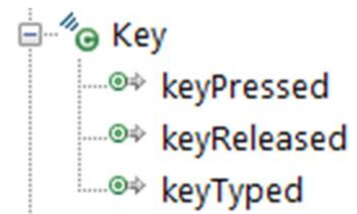
Exemple 2 : surclasser FOCUSADAPTER

```
public class Calculette    extends Frame
                           implements KeyListener, WindowListener
{
  // ...
  TextField saisieDollar, saisieEuros;

  Calculette()
  {
    // ...
    saisieDollar.addFocusListener(new MonFocusAdapter());
    // ...
  }

  class MonFocusAdapter extends FocusAdapter
  {
     public void focusGained(FocusEvent arg0)
     {
       saisieDollar.setText("");
       saisieEuros.setText("");
     }
  }
  // ...
}
```

Exemple 3 : définir l'écouteur dans une classe anonyme

```
public class Calculette    extends Frame
                           implements KeyListener, WindowListener
{
  // ...
  TextField saisieDollar, saisieEuros;

  Calculette()
  {
    // ...
    saisieDollar.addFocusListener(new FocusAdapter()
     {
       public void focusGained(FocusEvent arg0)
       {
          saisieDollar.setText("");
          saisieEuros.setText("");
       }
     }
    );
    // ...
  }
  // ...
}
```

Evénements liés au clavier



- Une touche est enfoncée : `keyPressed`
- Une touche relâchée : `keyReleased`
- Une touche frappée : `keyTyped`

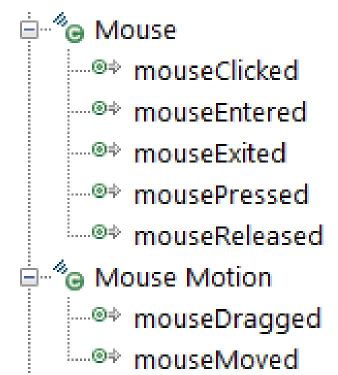
Abonnement

- `par addKeyListener`

Ecouteurs

```
public void keyPressed (KeyEvent e)
public void keyReleased(KeyEvent e)
public void keyTyped   (KeyEvent e)
```

Evénements liés au mulot : `Mouse`



- Clic bouton : `mouseClicked`
- Curseur entrant dans la zone active : `mouseEntered`
- Curseur sortant de la zone active : `mouseExited`
- Clic enfoncé : `mousePressed`
- Clic relâché : `mouseReleased`

Abonnement

- `par addMouseListener`

Ecouteurs

```
public void mouseClicked (MouseEvent e)
public void mouseEntered (MouseEvent e)
public void mouseExited  (MouseEvent e)
public void mousePressed (MouseEvent e)
public void mouseReleased(MouseEvent e)
```

Evénements liés déplacement du mulot : `Mouse Motion`

- Zone de donnée déplacée à l'aide de la souris : `mouseDragged`
- Souris déplacée : `mouseMoved`

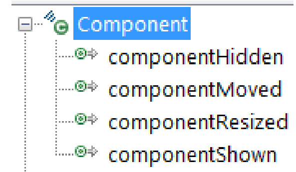
Abonnement

- `par addMouseMotionListener`

Ecouteurs

```
public void mouseDragged (MouseMotionEvent e)
public void mouseMoved   (MouseMotionEvent e)
```

Evénements déclenchés par les changements d'aspect d'un contrôle (`Component`)



- Passage du contrôle de l'état visible à l'état invisible : `componentHidden`
- Passage du contrôle de l'état invisible à l'état visible : `componentShown`
- Déplacement du contrôle : `componentMoved`
- Redimensionnement du contrôle : `componentResized`

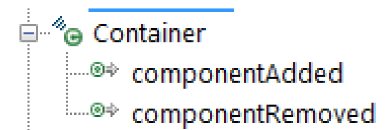
Abonnement

- par `addComponentListener`

Ecouteurs

```
public void componentHidden (ComponentEvent e)
public void componentShown  (ComponentEvent e)
public void componentMoved  (ComponentEvent e)
public void componentResized(ComponentEvent e)
```

Evénements déclenchés par l'ajout d'un contrôle (`Container`)



- Un contrôle est ajouté à la fenêtre : `componentAdded`
- Un contrôle est retiré de la fenêtre : `componentRemoved`

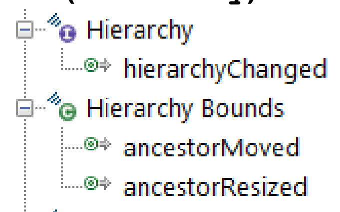
Abonnement

- par `addContainerListener`

Ecouteurs

```
public void componentAdded  (ContainerEvent e)
public void componentRemoved(ContainerEvent e)
```

Evénements déclenchés par le changement de position du contrôle sur l'axe Z (`Hierarchy`)



- L'ordre dans l'axe Z est modifié : `hierarchyChanged.`
- L'élément en position Z-1 est déplacé : `ancestorMoved.`
- L'élément en position Z-1 est retaillé : `ancestorResized.`

Abonnement pour `hierarchy`

- par `addHierarchyListener`

Abonnement pour `hierarchy Bounds`

- par `addHierarchy BoundsListener`

Ecouteurs

```
public void hierarchyChanged (HierarchyEvent e)
public void ancestorMoved    (HierarchyBoudsEvent e)
public void ancestorResized  (HierarchyBoudsEvent e)
```

CONTROLES DE TYPE TIMER

Un `Timer` est un contrôle non visuel permettant d'effectuer des actions de façon périodique. D'une façon générale les `Timers` sont à préférer aux `Threads` dans les applications graphiques (le `Timer` étant dans le même Thread que la pompe à messages).

- La méthode suivant créée un `Timer` de période 1000ms et défini son écouteur :

```
private Timer createTimer ()
{
  // A
  ActionListener action = new ActionListener ()
  {
      // Ecouteur anaoyme appelé à chaque tic du timer
      public void actionPerformed (ActionEvent event)
      {
        // Action exécutée à toutes les secondes
      }
  };
  return new Timer (1000, action); // 1000 peut être une variable entière
}
```

- Création du `Timer` :

```
//Zone attribut : définition de la référence
private Timer  timer;

//Dans une méthode
timer = createTimer();
```

- Accès au `Timer` :

```
//Dans des méthodes

// Démarrage
  timer.start();

// Arrêt
   timer.stop();

// Modification dynamique de la période
   timer.setDelay(100);

// A partir d'une scroll bar
   timer.setDelay(scrollbarDelay.getValue());
```

Page blanche

UTILISATION DE WINDOWS BUILDER SOUS ECLIPSE.

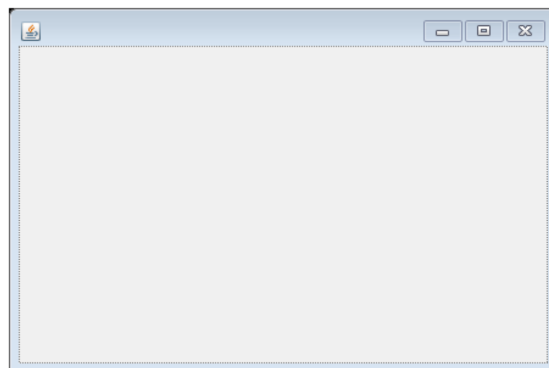
WindowBuilder est un *plugin* d'éclipse qui permet de créer facilement des IHMs. Le codé généré est très propre et facilement compréhensible. Il nécessite cependant de renseigner le code à la main.

CREATION D'UNE APPLICATION AVEC WB

1. Créer un projet de type Java
2. Créer une application WB : Files/new/Others/WindowBuilder/Swing Designer/Application Window

Le code généré est le suivant (la classe générée ici s'appelle `IhmMain`) :

```
import java.awt.EventQueue;
import javax.swing.JFrame;

public class IhmMain
{
  private JFrame frame;

  /**
   * Launch the application.
   */
  public static void main(String[] args)
  {
    EventQueue.invokeLater(new Runnable()
      {
        public void run()
        {
          try
          {
            IhmMain window = new IhmMain ();
            window.frame.setVisible(true);
          }
          catch (Exception e)
          {
            e.printStackTrace();
          }
        }
      });
  }

  /**
   * Create the application.
   */
  public IhmMain()
  {
    initialize();
  }

  /**
   * Initialize the contents of the frame.
   */
  private void initialize()
  {
    frame = new JFrame();
    frame.setBounds(100, 100, 450, 300);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```

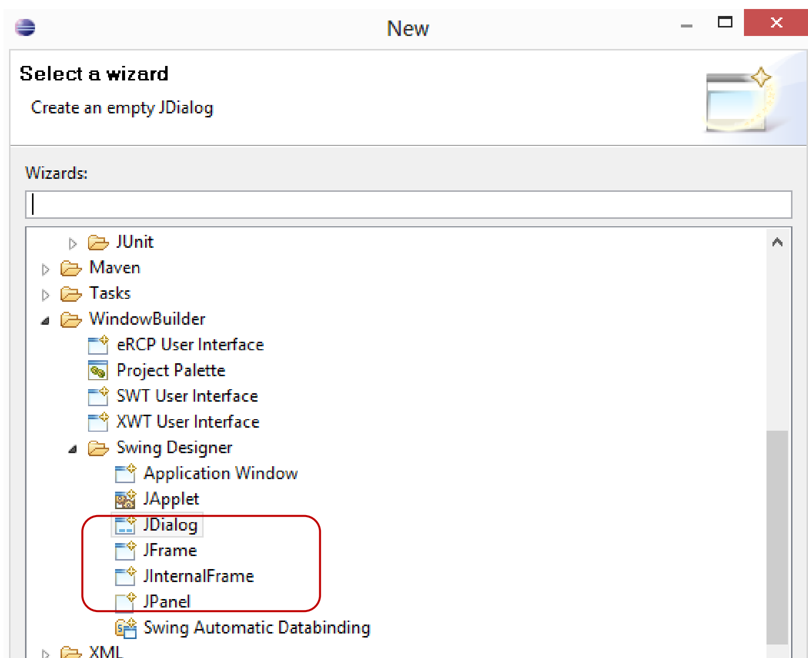


L'application peut être vue sous la forme de code : onglet *source*, ou sous forme graphique : onglet *design*.

L'ajout de code de façon manuelle doit se faire exclusivement après l'appel de la méthode `initialize`.

AJOUT D'UNE IHM DANS L'APPLICATION PRECEDENTE

1. Menu : Files/new/Others/WindowBuilder/Swing Designer/<choisir une des IHMs ci-dessous>



2. Supprimer la fonction main générée par WB
3. Instancier la classe et afficher l'objet par la méthode `setVisible`

Exemple : Ajouter une fenêtre Ihm2 de classe Frame à l'application précédente :

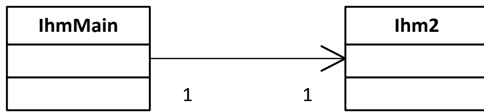
```
public class Ihm2 extends JFrame
{
  private JPanel contentPane;

  /**
   * Create the frame.
   */
  public Ihm2()
  {
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    setBounds(100, 100, 450, 300);
    contentPane = new JPanel();
    contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
    contentPane.setLayout(new BorderLayout(0, 0));
    setContentPane(contentPane);
  }

}
```

Pour afficher la fenêtre ajouter dans le constructeur de la classe `IhmMain` :

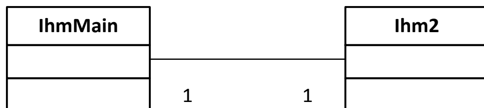
```
  public IhmMain()
  {
    initialize();
    Ihm2 frame = new Ihm2();
    //…
    frame.setVisible(true);
  }
```

Remarque :

- `frame` est déclaré localement au constructeur `IhmMain` et n'est donc plus accessible une fois l'objet `IhmMain` construit. Généralement ce type de référence est placé dans la zone attribut, de façon à pouvoir y accéder dans toutes les fonctions membres de la classe `IhmMain`.
- L'association réalisée est de type unidirectionnelle :



Pour réaliser une association bidirectionnelle :

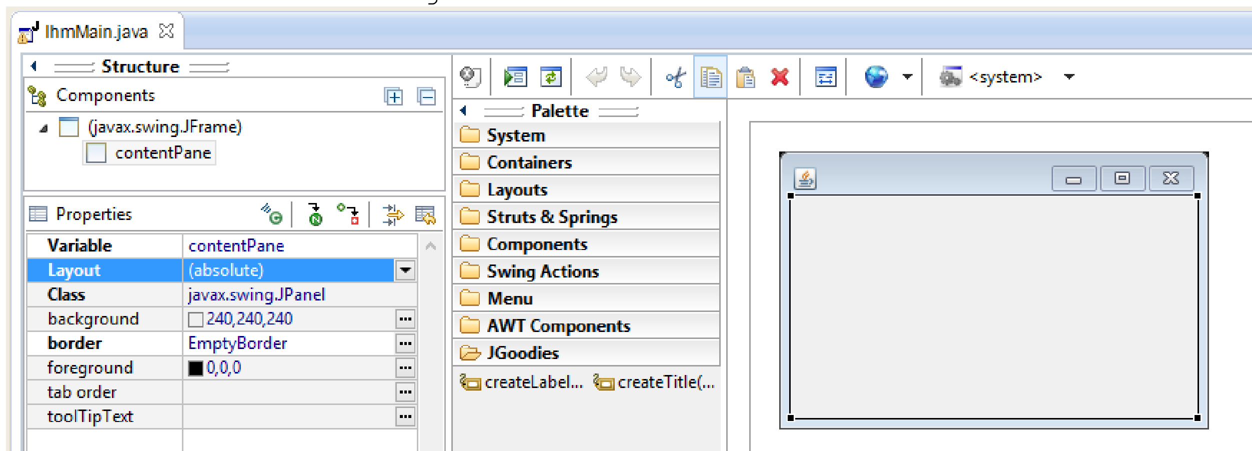


- Modifier le constructeur de la classe `Ihm2` de façon à accepter un paramètre de type `IhmMain` et mémoriser le paramètre dans un attribut.

```
public class Ihm2 extends JFrame
{
  private JPanel contentPane;
  private IhmMain ihmMain;

  /**
   * Create the frame.
   */
  public Ihm2(IhmMain ihmMain)
  {
    this.ihmMain = ihmMain ;

    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    // ...
  }
}
```

ENRICHISSEMENT DE LA FENETRE

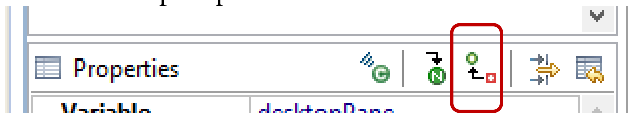
1. Ouvrir la classe en mode `Design`



La fenêtre présente les principaux composants graphiques :

- `System` permet principalement de définir le passage de contrôle en contrôle à l'aide de la touche de tabulation.
- `Containers` permet d'ajouter des panneaux (vues) à la fenêtre
- `Layouts` permet de définir le mode disposition des contrôles dans la fenêtre
- `Struts & Springs` permet de définir l'orientation des contrôles
- `Components` permet de choisir un contrôle de type Swing
- `Menus` permet de créer des menus
- `AWT Components` permet de choisir un contrôle de type AWT

2. Commencer par ajouter un ou plusieurs panneaux (si l'application utilise zones de fenêtre avec des *layouts* différents.
3. Définir le mode *layout* (généralement Absolute Layout) pour les panneaux utilisés.
4. Ajouter les autres composants dans les panneaux. Modifier la visibilité de l'attribut du contrôle si celui-ci doit être accessible depuis plusieurs méthodes.



5. Personnaliser les contrôles (couleur, police, etc..)
6. Ajouter les écouteurs aux contrôles (clic droit sur le contrôle ⇨ *Add event handler*
7. Définir le code des écouteurs. Exemple pour un bouton

```
JButton btnNewButton = new JButton("New button");

btnNewButton.addActionListener(new ActionListener() {

  public void actionPerformed(ActionEvent arg0) {

   // Code de l'écouteur à personnaliser

  }
});
```

LES APPLETTES

Une applette est un programme java exécuté dans un navigateur Web. Elle est téléchargée automatiquement depuis le site Web jusque dans la machine exécutant le navigateur.

CREATION

Pour créer une applette il faut :

1. Dériver la classe principale de l'application de la classe Applet.
2. Ne pas instancier d'objets Frame, un applet utilise la fenêtre (frame) du navigateur
3. Définir une fonction `init()` (non statique) à la place de main instanciant la classe principale.
4. Définir le code de lancement de l'applette dans la page HTML.
   Par exemple pour une classe `CalcComplexGraphApplet` on définit :

```
<HTLM>
 <BODY>
  <p align="center">
    <APPLET CODE="CalcComplexGraphApplet" WIDTH=200 HEIGHT=160>
    </APPLET>
  </p>
 </BODY>
</HTLM>
```

SECURITE DES APPLETTES

Une applette ne peut ni accéder aux périphériques de stockage locaux, ni au serveur dont il est l'origine.

Page blanche

LES APPLICATIONS JAVA WEBSTART

Une application Java WebStart est une application chargée sur la machine cliente depuis le serveur distant.
Pour accéder aux périphériques de stockage locaux et au serveur dont il est l'origine, l'application doit être signée.

CREATION D'UNE APPLICATION JAVA WEBSTART

a) A partir d'une application java classique (console, ou fenêtrée) faire une archive `JAR` (voir en début de ce document)
b) Créer un fichier de lancement. C'est un fichier texte d'extension `jnlp`. Par exemple pour le QCM de type QUIZ on définit le fichier suivant `quiz.jnlp` contenant (en gras les lignes à paramétrer) :

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <jnlp spec="1.0"
3.   codebase="http://daniel.tschirhart.free.fr/qcmprog"
4.   href="quiz.jnlp">
5.   <information>
6.    <title>Quiz</title>
7.      <vendor>Daniel Tschirhart</vendor>
8.      <icon href=" "/>
9.      <offline-notallowed/>
10.       </information>
11.       <resources>
12.         <jar href="squiz.jar"/>
13.         <j2se version="1.6+"
14.         href="http://java.sun.com/products/autodl/j2se"/>
15.       </resources>
16.       <security>
17.         <all-permissions/>
18.       </security>
19.       <application-desc main-class="s"/>
20.     </jnlp>
```

Ligne 3 emplacement du fichier `JAR` sur le serveur
Ligne 4 nom du fichier `jnlp`
Ligne 7 nom de l'éditeur
Ligne 12 nom du fichier `jar`
Ligne 13 version minimale pour exécuter l'application
Ligne 14 si la machine java n'est pas présente dans la version voulue, adresse de téléchargement de JRE
Ligne 17 autorisation accordée à l'application
Ligne 19 nom de la classe principale

c) Pour être certain que l'application ne sera pas modifiée par un programme maveillant, il faut générer une clé pour signer l'application. L'opération est réalisée avec l'utilitaire `keytool` fourni avec le JDK :

```
keytool -genkey -validity 3650 -alias signature -keystore signatureStore
```

La clé possède ici une durée de validité de 10 ans (3650 jours), elle est mémorisée dans le magasin `signatureStore.` Cette clé peut signer un nombre illimité d'applications

d) Signer l'application avec `jarsigner` fourni avec le JDK

```
Jarsigner -keystore SignatureStore -signedjar squiz.jar quizpro.jar signature
```

On précise le magasin où est la clé est mémorisée, le nom du fichier une fois signé (ici `squiz.jar`), le nom de l'application à signer (ici `quizpro.jar`).

e) Copier sur le site Web le fichier `jnlp` et le fichier jar signé. Préciser le lien du fichier `jnlp` dans la page Web.

Page blanche

JAVA ET LA PROGRAMMATION MULTITACHES

LES TACHES (THREAD)

DEFINITION

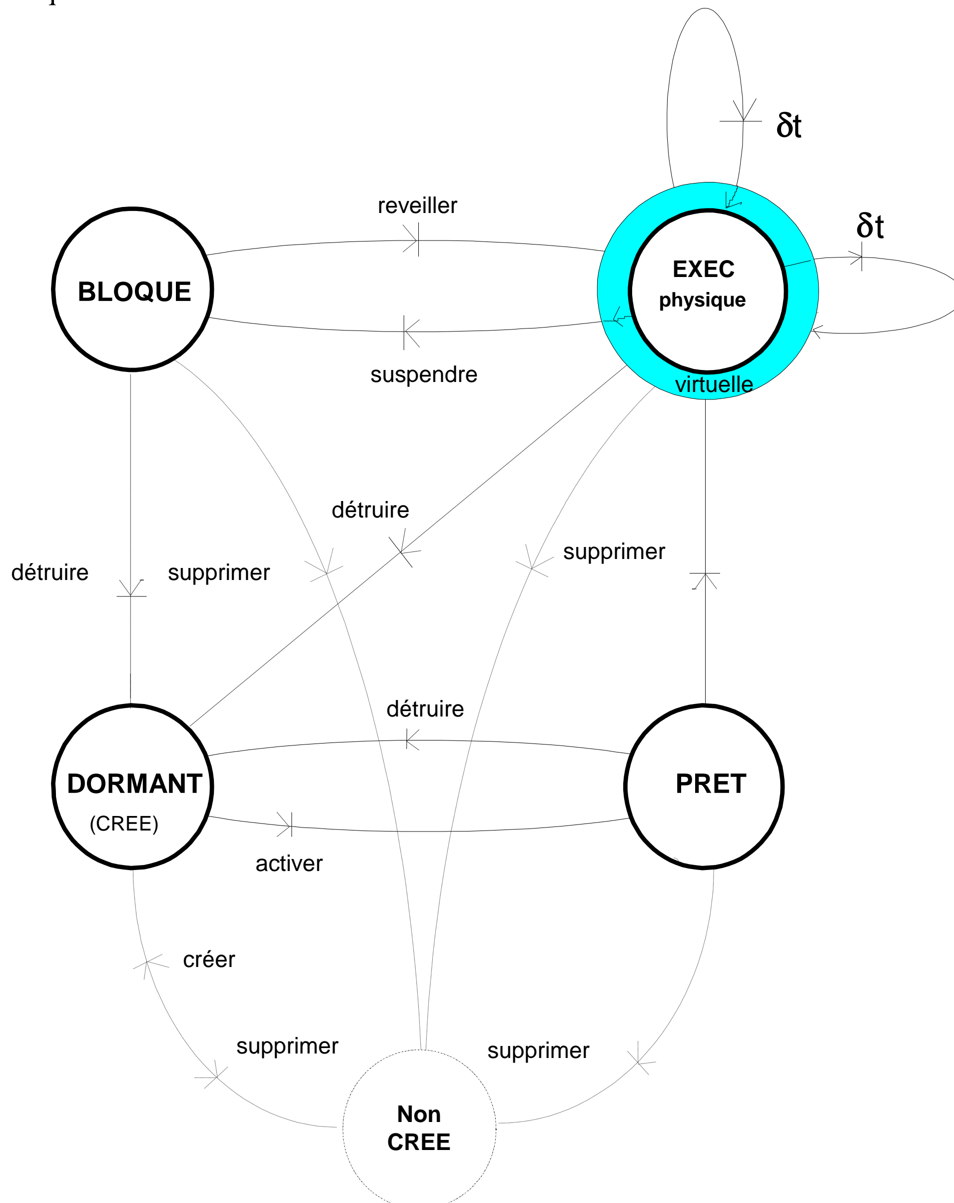
La programmation multitâche ou programmation parallèle vise à utiliser au mieux les ressources matérielles d'un système programmable.

Une tâche (ou Thread) est une fonction exécutée de façon concurrente avec d'autres fonctions. Une tâche possède son jeu de variables locales mais partage ses attributs avec les autres tâches.

Les tâches sont gérées nativement par l'exécutif multitâche du système d'exploitation, Java ne fait qu'encapsuler les primitives de gestion des tâches. Les performences multitâches Java dépendent entièrement des performences multitâche du système d'exploitation hôte.

ETATS D'UNE TACHE

A un instant donné, chaque tâche se trouve dans des états suivant :



Le passage d'un état à un autre s'effectue généralement à l'aide d'appel de requêtes présentes dans l'objet Thread.

Les différents états possèdent les propriétés suivantes :

ETAT NON CREE

La tâche inconnue, aucune ressource ne lui est allouée.

ETAT DORMANT OU CREE

Tâche connue de l'exécutif. Un identificateur, ainsi qu'une zone de pile est attribué à la tâche. La tâche reste dans cet état tant qu'une requête d'activation ne la fait pas évoluer.

ETAT PRET OU ELIGIBLE

Une tâche à l'état prêt est candidate pour l'exécution. Son lancement ne dépend que de sa priorité par rapport aux autres tâches éligibles ou en cours d'exécution. Lorsque celle-ci devient prioritaire, l'ordonnanceur lui attribue le processeur. L'exécution de la tâche commence alors à son début.

ETAT EN EXECUTION

Une tâche en exécution est une tâche en possession du processeur. Dans un système monoprocesseur, plusieurs tâches de même priorité peuvent être en exécution mais une seule tâche possède le processeur phisique, chaque tâche se voyant attribuer un quantum de temps processeur. Une tâche reste dans cet état tant qu'elle est prioritaire et qu'elle ne n'exécute pas une requête de suspension directe ou indirecte.

ETAT BLOQUE OU SUSPENDU

Une tâche à l'état bloqué ou à l'état suspendu ne possède plus le processeur et son état est sauvegardé en vue de sa reprise ultérieure.

TRANSITIONS ENTRE ETATS D'UNE TACHE

Le franchissement des transitions et la conséquence d'appels directs ou indirects aux primitives de l'exécutif.

CREATION D'UNE TACHE (CREE)

Créer une tâche consiste à la faire connaître de l'exécutif : fournir une adresse de début, une taille de pile et un niveau de priorité. En retour le noyau fournit un identificateur permettant de référencer la tâche.

La détermination de la taille de pile n'est pas chose aisée. La pile est utilisée pour loger les variables locales et les adresses de retour lors des appels de fonctions. Elle sert également à mémoriser le contexte du processeur lors des commutations de tâche. Une pile insuffisante est souvent la cause de dysfonctionnement mystérieux.

ACTIVATION D'UNE TACHE (ACTIVE)

L'activation d'une tâche correspond à une demande d'exécution. L'activation peut être immédiate, différée ou cyclique.

L'activation immédiate consiste à faire passer une tâche de l'état dormant à l'état prêt. En aucun cas elle ne correspond à l'appel direct de la fonction spécifiant la tâche : l'instant du lancement d'une tâche est dévolu à l'ordonnanceur.

Les demandes d'activation sont généralement mémorisées. Ainsi à trois demandes successives d'activations correspondent trois cycles (on suppose que la tâche de termine) :

$$DORMANT \rightarrow PRET \rightarrow EXEC \rightarrow DORMANT.$$

ATTRIBUTION DU PROCESSEUR A LA TACHE

La tâche est lancée depuis son début. Son début d'exécution n'est pas forcément immédiat : si d'autres tâches (de même priorité) sont déjà dans cet état, la tâche attendra son tour suivant le mécanisme du tourniquet. Cette transition n'est pas maîtrisée par le programmeur.

SUSPENSION D'UNE TACHE

La tâche est stoppée et son contexte est enregistré de façon à pouvoir reprendre exactement à l'endroit ou elle s'est arrêtée. La suspension de la tâche peut être explicite ou être le résultat d'un blocage lors d'une opération Entrée/Sortie.

REPRISE D'UNE TACHE BLOQUEE

Cette transition n'est pas contrôlée directement. Lorsque toutes les conditions sont réunies pour continuer une tâche (priorité maximale de la tâche par rapport aux autres, disparition des conditions de blocage) le contexte de la tâche est rechargé et celle-ci est relancée à l'endroit où elle s'était arrêtée.

FIN D'UNE TACHE OU DESTRUCTION D'UNE TACHE

Une tâche terminée ou détruite est replacée à l'état prêt. La fin d'une tâche peut être obtenue par un appel explicite au noyau ou implicite lors de la fin de la tâche. La destruction explicite d'une tâche est une opération très fortement déconseillée car amener un blocage de l'ensemble du programme.

SUPPRESSION D'UNE TACHE

La suppression explicite d'une tâche est une opération très fortement déconseillée car amener un blocage de l'ensemble du programme.

ATTRIBUTION DU PROCESSEUR PHYSIQUE

Cette transition a lieu tous les quanta pour une tâche à l'état *EXEC*. Elle n'est pas maîtrisable directement.

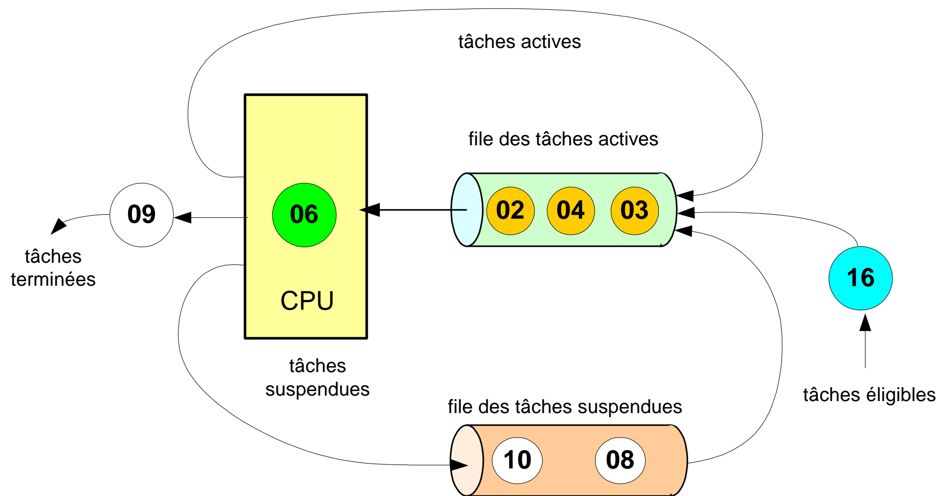
ABANDON DU PROCESSEUR PHYSIQUE

Cette transition a lieu chaque fois qu'une tâche a utilisé le processeur physique durant un quantum.

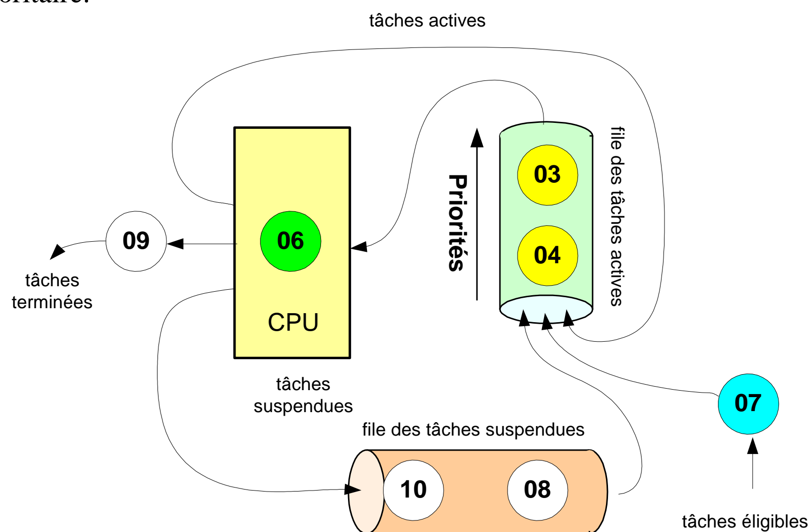
POLITIQUES D'ORDONNANCEMENT

La politique d'ordonnancement définit le critère utilisé par le noyau pour élire une tâche. Diverses politiques d'ordonnancement sont utilisées :

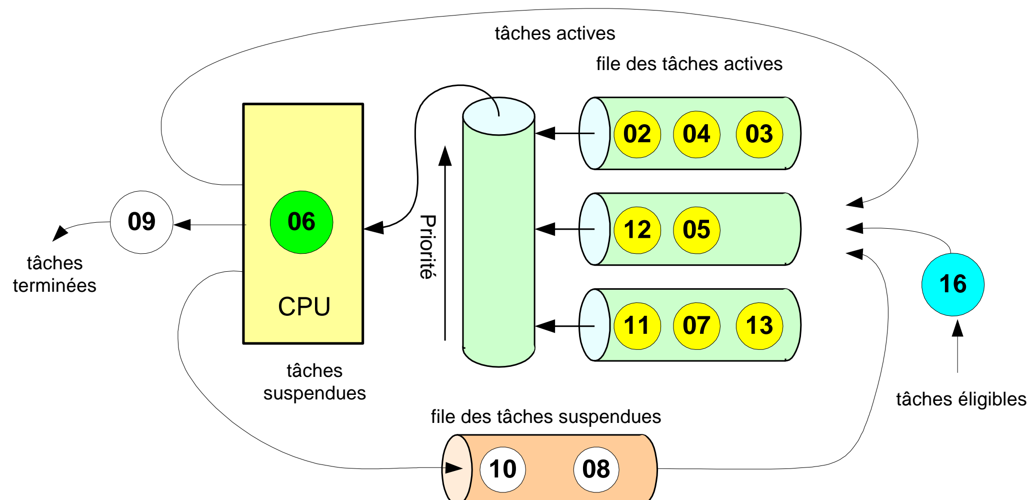
- Ordonnancement circulaire ou tourniquet : Chaque tâche possède la même priorité. Les tâches se partagent le processeur sans affinité particulière.



- Ordonnancement à priorité fixe : Chaque tâche possède un niveau de priorité différent. La tâche exécutée est toujours la plus prioritaire.



- Ordonnancement à classe de priorités : Combinaison des deux politiques précédentes. Les tâches sont groupées par classe de priorité au sein des quelles l'ordonnancement est de type circulaire. Ce système présente le maximum de souplesse.



- Ordonnancement par priorité avec modification dynamique de la priorité en fonction de l'âge de la tâche. Cette politique fait perdre progressivement la priorité aux tâches : les tâches de courte durée sont toujours exécutées en premier. Cette politique substitue au contrôle des tâches exercées par le programmeur, par celui exercé par le noyau.

LES TACHES DANS JAVA

Avant d'utiliser les tâches (`Threads`), demander-vous si un objet de type `Timer` ne pourrai pas faire l'affaire.

Dans une application graphique `Thread` ne peut modifier un contrôle que si celui-ci fait partie de ce `Thread.`

PACKAGE

Java encapsule les tâches dans la classe Thread du package `java.thread`

PRIORITES DES TACHES

La priorité d'une tâche est définie par une valeur entière constante comprise entre les constantes statiques `MAX_PRIORITY et MIN_PRIORITY. NORM_PRIORITY` correspond à la priorité médiane.

Java s'appuie sur le système d'exploitation hôte pour attribuer la priorité aux tâches. Si le système d'exploitation n'implémente pas un mécanisme d'ordonnancement à priorités (UNIX, Linux) l'attribution des priorités aux taches est ignorée.

CREATION DES TACHES

Deux méthodes sont utilisables :

- Sous classer la classe **`Thread`** et surcharger la méthode **`run.`**
  Cette méthode ne convient pas pour les applications dérivant déjà une classe (généralement les applications graphiques).
- Implémenter l'interface **`Runnable`**

Constructeurs de la classe Thread

- `Thread()`
  Constructeur par défaut. Créée un nouvel objet Thread et lui attribue un nom : `"Thread-"+n n:0..`
- `Thread(String name)`
  Idem au constructeur par défaut, permet de nommer le Thread.
- `Thread(Runnable target)`
  Créée un nouvel objet Thread. Target est un objet qui doit implementer la méthode run qui sera invoquée comme Thread
- `Thread(Runnable target, String name)`
  Idem au constructeur précédent, permet de nommer le Thread.

Création des tâches par sous classement de la classe Thread

```
public class ThreadSousClassé extends Thread
{
  public void run()
  {
    System.out.println(this.getName()+ " en éxécution");
    try
    {
        Thread.sleep(1000);
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }
    System.out.println(this.getName()+ " fin");
  }

  public static void main(String[] args)
  {
    ThreadSousClassé t1 = new ThreadSousClassé();
    t1.start();
    ThreadSousClassé t2 = new ThreadSousClassé();
    t2.start();
  }
}
```

Création des tâches par implémentation de l'interface `runnable`

```
public class Thread2 implements Runnable
{

  public void run()
  {
    System.out.println(Thread.currentThread().getName()+ " en éxécution");
    try
    {
        Thread.sleep(1000);
    }
    catch (InterruptedException e)
    {
        e.printStackTrace();
    }
    System.out.println(Thread.currentThread().getName()+ " fin");
  }

  public static void main(String[] args)
  {
    Thread2 t1 = new Thread2();
    new Thread(t1).start();
    Thread2 t2 = new Thread2();
    new Thread(t2).start();
  }
}
```

Principales méthodes

- `static Thread currentThread()`
  Retourne la référence du thread courant.
- `String getName()`
  Retourne le nom de la tâche.
- `int getPriority()`
  Retourne la priorité de la tâche.
- `void setPriority()`
  Fixe la priorité de la tâche.
- `static Thread currentThread()`
  Retourne la référence du thread courant.
- `void interrupt()`
  Relance une tâche suspendue (en sommeil)
- `static void sleep(long ms)`
  Suspend la tâche durant ms millisecondes

PROBLEMES POSE PAR LES ACCES CONCURRENTS

REENTRANCE DES FONCTIONS

Une fonction est réentrante si elle s'exécute correctement lorsqu'elle est appelée simultanément par plusieurs tâches. Une fonction effectuant des récusions directes ou indirectes doit être réentrante. Une fonction réentrante ne peut modifier des variables situées à des emplacements fixe de la mémoire (variables statiques)..

Voici un exemple de fonction non réentrante:

```
static int fac;
int factorielle(int n)
{
  fac = 1;
  while ( n != 0 ) {
    fac = fac * n;
    n--;
  }
  return ( fac );
}
```

```
void A(void)
{
  ...
  factorielle(5);
  ...
}
```

```
void B(void)
{
  ...
  factorielle(3);
  ...
}
```

Deux taches A et B effectuent simultanément l'appel à *factorielle*. La fonction *factorielle* n'est pas réentrante du fait que le variable *fac* est de type *static*, donc situé à un emplacement fixe de la mémoire.
Supposons que les tâches A et B soient lancés simultanément et que la tâche *B* s'exécute en premier et effectue *fac = 1*. Supposons que la tâche *A* prend possession du processeur juste après cette affectation et réalise le calcul *factorielle(5).* Lorsque la tâche A se termine et le calcul de *factorielle(5)* a donné à *fac* la valeur 120 (!5). Le calcul reprend ensuite dans B avec *fac = 120* et fournit évidemment un résultat faux pour *factorielle(3).*
Pour rendre la fonction *factorielle* réentrante il suffit de déclarer *fac* dans le corps de la fonction. L'appel de *factorielle* par une tâche créé une nouvelle variable *fac* dans la pile de la tâche. Comme chaque tâche possède sa propre pile, la fonction *factorielle* peut travailler simultanément avec plusieurs tâches.

L'exécution d'une fonction non réentrante n'a ici pour conséquence qu'une erreur de calcul. Il en serait tout autrement si la fonction doit prendre des décisions (test sur la valeur d’une variable à accès concurrent).
Une fonction réentrante ne doit pas appeler une fonction non réentrante sous peine de devenir à son tour non réentrante.
Une fonction doit être réentrante si elle est susceptible d'être appelée simultanément par plusieurs tâches.
Sections critiques.

SECTIONS CRITIQUES

Une section critique est une zone de programme où se produit un accès concurrent et dont le résultat des traitements et fonction de l’ordonnancement des tâches ou des processus.
Un programme multitâche ou multiprocessus doit absolument verrouiller l’accès aux sections critiques en en autorisant l’accès à une seule tâche à la fois. La multiplicité des sections critiques produit à un code peu efficient du fait des appels au noyau multitâche destiné à verrouiller les sections critiques.

OBJETS UTILISES POUR LE VERROUILLAGE DES SECTION CRITIQUES

Sémaphores

Les sémaphores ont été proposés pour la première fois en 1965 par E. W. Dijkstra. Dijkstra propose de compter le nombre de tâches endormies ou les réveils en attente à l'aide d'une variable appelée sémaphore.

Un sémaphore est constitué d'un entier *s* signé pouvant prendre des valeurs positives et négatives ou nulles et d'une file d'attente mémorisant les contextes des tâches en attente. L'accès à la variable *s* est effectué dans une section critique verrouillée.

Deux primitives permettent de manipuler le sémaphore : P(s) et V(s).

```
P(s)                                                          :
  s <- s-1
  Si s < 0 Alors
    bloquer la tâche courante
    mémoriser l'identificateur de la tâche courante dans la file f(s)
  Fin

V(s)                                                          :
  s <- s+1
  Si s ≤ 0 Alors
    relancer de la tâche en tête de la file f(s)
  Fin
```

La modification et le test de *s* (zone entourée) est une section critique codé à l'aide d'instructions spéciales (TestAndSet) du processeur permettant d'exécuter ce code de façon atomique. Ces instructions sont des instructions exécutables uniquement en mode priviligés.

SEMAPHORES DANS JAVA

Les sémaphores sont définis dans le package `java.util.concurrent`

`Classe Semaphore`

- `Semaphore(int init)` : constructeur spécifiant la valeur initiale du sémaphore.
- `void acquire() : opération P(s)`
- `void release() : opération V(s)`
-

VERROUS

Un verrou est un objet permettant de verrouiller l'accès à une zone de code.

VERROUS DANS JAVA

Les verrous sont définis dans le package `java.util.concurrent.locks`

L'interface `Lock` fourni les méthodes permettant de verrouiller une section critique. Ces méthodes sont implémentées dans la classe `ReentrantLock`.

Principales fonctions membres de l'interface `Lock`

`Interface Lock`

- `lock()` : verrouille le code qui suit
- `unlock()` : déverrouille la section de code précédente
-

Verrouillage des sections critiques à l'aide des sémaphores

Il faut initialiser le sémaphore à 1.

```
static Semaphore mutex = new Semaphore(1);
// ...
// Début de section critique
try
{
  mutex.acquire();
  // Section critique
}
catch (InterruptedException e)
{
  e.printStackTrace();
}
// Fin section critique
mutex.release();
```

Verrouillage des sections critiques à l'aide des verrous

```
  static Lock    verrou = new ReentrantLock();
  // ...
  // Début de section critique
  verrou.lock();
  // section critique
  verrou.unlock();
  // Fin de section critique
```

MODELES D'ECHANGE DE DONNEE

L'exclusion mutuelle des sections critiques autorise un mécanisme d'échange rudimentaire. Dans ce type d'échange, il n'y a aucune synchronisation entre les protagonistes. Le seul service fourni par l'exclusion mutuelle est la garantie qu'une donnée est lue ou écrite de façon atomique, ou que dans l'intervalle de temps entre la lecture de la donnée et son test, celle-ci n'a pas été modifiée.
Lorsqu'on désire synchroniser les échanges entre protagonistes deux modèles sont utilisés :

- Le modèle producteur-consommateur
- Le modèle lecteur-rédacteur

MODELE PRODUCTEUR-CONSOMMATEUR

Ce modèle est la base de nombreux échanges de données que ce soit au cœur d'un système d'exploitation ou au sein des applications. Son importance est telle que la plupart des noyaux fournissent directement ce type d'échange (tubes, messages, ...).
Les règles de production et de consommation sont les suivantes :

- Une donnée ne peut être consommée que si elle a été produite,
- Une donnée ne peut être produite que si la donnée précédente a été consommée,

Il est résulte qu'une même donnée ne peut être consommée ou produite plusieurs fois.
L'implémentation de ce mécanisme est basée sur la propriété des sémaphores.
On distingue deux sous modèles : le modèle ne gérant pas de stocks et le modèle gérant des stocks

Producteur consommateur sans stocks

Le pseudo code ci-dessous illustre ce modèle. Les taches `producteur` et `consommateur` s'exécutent de façon concurrentes.

```
DONNEE d;
Semaphore prod(1);
Semaphore cons(0);

Tache producteur()
{
  while (true)
  {
     P(prod);
     d = produire();
     V(cons);
  }
}

Tache consommateur()
{
  while (true)
  {
     P(cons);
     consommer(d);
     V(prod);
  }
}
```

Exemple utilisant les sémaphores Java

```
import java.util.concurrent.*;

class ProdCons implements Runnable
{
  static Semaphore prod  = new Semaphore(1);
  static Semaphore conso = new Semaphore(0);
  String           msg;
  static int       donnée;  // donnée échangée

  public ProdCons(String msg)
  {
    this.msg = msg;
  }
```

```
public void run()
  {
    System.out.println(Thread.currentThread().getName() + " : début d'exécution...");
    // Début de section critique
    try
    {
      if (msg.equals("P"))
      {
        for (int i = 0; i < 1000; i++)
        {
          prod.acquire();
          donnée = i;
          conso.release();
        }
      }
      else
      {
        for (int i = 0; i < 1000; i++)
        {
          conso.acquire();
          System.out.println(donnée);
          prod.release();
        }
      }
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
    System.out.println(Thread.currentThread().getName() + " : fin d'exécution.");
  }

  static public void main(String[] args)
  {
    System.out.println("Producteur consommateur sans stocks");
    ProdCons thread1 = new ProdCons("P");
    new Thread(thread1, "Producteur").start();
    ProdCons thread2 = new ProdCons("C");
    new Thread(thread2, "Consommateur").start();
  }
}
```

Producteur consommateur avec stocks

Le pseudo code ci-dessous illustre ce modèle. Les taches `producteur` et `consommateur` s'exécutent de façon concurrentes. Les données sont mémorisées dans une fifo comportant 100 emplacements. Le sémaphore `prod` gère la place disponible dans la fifo.

```
Fifo file(100);
Semaphore prod(100);
Semaphore cons(0);
```

```
Tache producteur()
{
  while (true)
  {
     P(prod);
     file.add(produire());
     V(cons);
  }
}
```

```
Tache consommateur()
{
  while (true)
  {
     P(cons);
     consommer(file.get());
     V(prod);
  }
}
```

Exemple utilisant les sémaphores Java

```
import java.util.*;
import java.util.concurrent.*;

public class ProdConsStocks implements Runnable
{
  static Semaphore prod  = new Semaphore(100);
  static Semaphore conso = new Semaphore(0);
  String           msg;
  // La fifo construite avec une liste sure
  static Queue<Integer> fifo = new ConcurrentLinkedQueue<Integer>();

  public ProdConsStocks(String msg)
  {
    this.msg = msg;
  }
  // La function run contient à la fois le producteur et le consommateur par la valeur de l'attribut msg
  public void run()
  {
    System.out.println(Thread.currentThread().getName() + " : début d'exécution...");
    try
    {
      if (msg.equals("P")) // le producteur
      {
        for (int i = 0; i < 1000; i++)
        {
          prod.acquire();
          fifo.offer(i);
          conso.release();
        }
      }
      else                 // le consommateur
      {
        for (int i = 0; i < 1000; i++)
        {
          conso.acquire();
          System.out.println(fifo.poll());
          prod.release();
        }
      }
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
    System.out.println(Thread.currentThread().getName() + " : fin d'exécution.");
  }
```

```
  public static void main(String[] args)
  {
    System.out.println("Producteur consommateur avec stocks");
    ProdConsStocks t1 = new ProdConsStocks("P");
    new Thread(t1, "Producteur").start();
    ProdConsStocks t2 = new ProdConsStocks("C");
    new Thread(t2, "Consommateur").start();
  }
}
```

Implémentation du modèle producteur consommateur dans Java

Vu l'importance du modèle producteur consommateur, celui-ci est directement implémenté dans Java dans les structures de données fondamentale (SDF) :
`ArrayBlockingQueue, LinkedBlockingQueue, PriorityBlockingQueue`

- `ArrayBlockingQueue` est une fifo construite avec un vecteur.
- `LinkedBlockingQueue` est une fifo construite avec une liste liée.
- `PriorityBlockingQueue` est une fifo à priorité.

Toutes ces SDF implémentent l'interface `Queue`.
L'exemple précédent peut s'écrire :

```
import java.util.*;
import java.util.concurrent.*;

public class PipeSample implements Runnable
{
  String                msg;
  static Queue<Integer> queue = new LinkedBlockingQueue<Integer>();

  public PipeSample(String msg)
  {
    this.msg = msg;
  }

  public void run()
  {
    System.out.println(Thread.currentThread().getName() + " : début d'exécution...");
    if (msg.equals("P"))
    {
      for (int i = 0; i < 200; i++)
      {
        queue.offer(i);
      }
    }
    else
    {
      for (int i = 0; i < 200; i++)
      {
        System.out.println(queue.poll());
      }
    }
    System.out.println(Thread.currentThread().getName() + " : fin d'exécution.");
  }

  public static void main(String[] args)
  {
    System.out.println("Producteur consommateur avec stocks");
    PipeSample t1 = new PipeSample("P");
    new Thread(t1, "Producteur").start();
    PipeSample t2 = new PipeSample("C");
    new Thread(t2, "Consommateur").start();
  }
}
```

MODELE LECTEUR - REDACTEUR

Contrairement au modèle producteur - consommateur, le modèle lecteur - rédacteur fait appel à plusieurs rédacteurs et plusieurs lecteurs simultanés. Le modèle lecteur - rédacteur est le modèle de base dans les applications de type client serveur. Il se conforme aux règles suivantes :

- Une donnée peut être lue simultanément par plusieurs lecteurs sans consommation de celle-ci.
- Un seul rédacteur à la fois peut modifier une donnée.
- Lorsqu'un rédacteur est actif, tous les lecteurs sont interdits de lecture.
- Lorsqu'un ou plusieurs lecteurs sont actifs tous les rédacteurs sont interdits d'écriture.

Le modèle lecteur – rédacteur n'est généralement jamais directement implémenté, la programmation de ce modèle sort du cadre de ce cours.

SYNCHRONISATIONS DES TACHES

Problème posé

Il s'agit de synchroniser plusieurs tâches ou un processus entre eux.

Deux types de mécanisme peuvent être utilisés :

- Mécanismes de synchronisation directs : ils désignent directement la tâche à synchroniser
- Mécanismes de synchronisation indirects : désignent indirectement la tâche à synchroniser à travers un objet de synchronisation.

MECANISMES DE SYNCHRONISATION DIRECTS

La mise en œuvre de ce mécanisme est réalisée par les fonctions de bas niveau et sort du cadre de ce cours

MECANISMES DE SYNCHRONISATION INDIRECTS

Ces mécanismes utilisent des objets du noyau multitâche par l'intermédiaire desquels s'effectue la synchronisation effective. Les objets utilisables pour cet usage sont les sémaphores, les variables événements, les variables rendez-vous.

SYNCHRONISATION A L'AIDE DE SEMAPHORES

Les sémaphores sont utilisés de façon suivante :

1. Une tâche (logicielle) lance une opération E/S puis se suspend en appelant la primitive *P(s)* avec *s* initialisé à 0.
2. Lorsque l'opération E/S est terminée, le périphérique concerné lance une tâche matérielle (programme d'interruption) qui effectue la primitive *V(s)*. La tâche initiatrice le l'opération E/S se trouve relancée et récupère la donnée.

SYNCHRONISATION A L'AIDE DE RENDEZ-VOUS

La synchronisation par rendez-vous permet de résoudre de façon simple la synchronisation d'un nombre connu de tâches.

La puissance de mécanisme fait qu'il est incorporé de façon native dans le langage ADA et existe également dans Java.

Les rendez-vous permettent de résoudre de nombreux problèmes de synchronisation.

La classe `CyclicBarrier` implémente la technique du rendez-vous. Supposons par exemple, que plusieurs tâches travaillent sur certaines parties d'un calcul. Lorsque toutes les parties sont prêtes, il faut combiner les résultats. Les tâches doivent donc se donner rendez-vous.

Le mécanisme à mettre en œuvre est le suivant :

1. Initier le rendez-vous avec le nombre de tâches.
   `CyclicBarrier rdv = new CyclicBarrier(nTask);`
   On peut également spécifier une action a effectuer lorsque le rendez-vous est réalisé
   `CyclicBarrier rdv = new CyclicBarrier(nTask, Action); //` Action est une classe implémenant l'interface `Runnable`.
2. Poser le rendez-vous :
   `rdv.await();`
   On peut borner la durée du rendez-vous :
   `rdv.await(100.TimeUnit.MILLISECONDS);`
3. Lorsque le rendez-vous est effectué l'objet `CyclicBarrier` peut être réutilisé.

Exemple 1

```
import java.util.concurrent.*;

public class RdvSample implements Runnable
{
  String              msg;
  static CyclicBarrier rdv = new CyclicBarrier(3);

  public RdvSample(String msg)
  {
    this.msg = msg;
  }

  public void run()
  {
    System.out.println(Thread.currentThread().getName() + " : début d'exécution...");
    try
    {
      Thread.sleep(Integer.parseInt(msg));
      rdv.await();
    }
    catch (BrokenBarrierException e)
    {
      e.printStackTrace();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
    System.out.println(Thread.currentThread().getName() + " : au rendez-vous.");
  }

  public static void main(String[] args)
  {
    System.out.println("Exemple de rendez-vous");
    RdvSample t1 = new RdvSample("1000");
    new Thread(t1, "Tache 1").start();
    RdvSample t2 = new RdvSample("2000");
    new Thread(t2, "Tache 2").start();
    RdvSample t3 = new RdvSample("3000");
    new Thread(t3, "Tache 3").start();
  }
}
```

Exemple 2

```
import java.util.concurrent.*;

class MsgRdv implements Runnable
{
  public void run()
  {
    System.out.println("Action effectuée (par un seul thread)");
  }
}

public class RdvSample2 implements Runnable
{
  String                msg;
  static CyclicBarrier rdv = new CyclicBarrier(3, new MsgRdv());

  public RdvSample2(String msg)
  {
    this.msg = msg;
  }

  public void run()
  {
    System.out.println(Thread.currentThread().getName() + " : début d'exécution...");
    try
    {
      Thread.sleep(Integer.parseInt(msg));
      rdv.await();
    }
    catch (BrokenBarrierException e)
    {
      e.printStackTrace();
    }
    catch (InterruptedException e)
    {
      e.printStackTrace();
    }
    System.out.println(Thread.currentThread().getName() + " : au rendez-vous.");
  }

  public static void main(String[] args)
  {
    System.out.println("Exemple de rendez-vous");
    RdvSample2 t1 = new RdvSample2("1000");
    new Thread(t1, "Tache 1").start();
    RdvSample2 t2 = new RdvSample2("2000");
    new Thread(t2, "Tache 2").start();
    RdvSample2 t3 = new RdvSample2("3000");
    new Thread(t3, "Tache 3").start();
  }
}
```

LA PROGRAMMATION RESEAU

SERVEURS ET CLIENTS

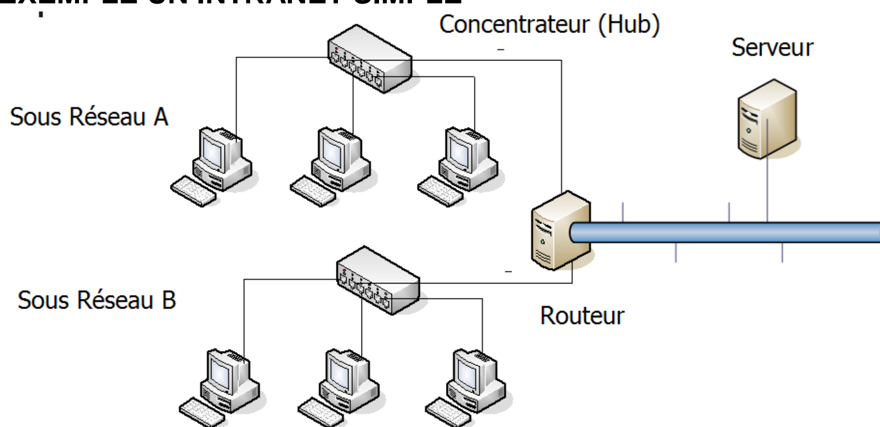
Un serveur est un ordinateur exécutant un programme qui attend des requêtes d'un ou plusieurs clients.
Le serveur est généralement relié au client par l'intermédiaire d'un réseau, mais peut également résider sur la même machine que le client.
Le type de programme exécuté par le serveur définit le type de requête que peut effectuer le client. Exemple :

- Serveur Web : accepte des requêtes http
- Serveur FTP : accepte des requêtes ftp
- Serveur de fichier : accepte des requêtes de lecture et d'écriture de fichiers
- Serveur de base de données : accepte des requêtes d'interrogation de base de données (serveur SQL : requêtes SQL)
- ...

Les requêtes sont transmises au serveur par l'intermédiaire d'un protocole réseau.

- Les serveurs de fichiers acceptent différents protocoles.
- Les serveurs reliés par le réseau Internet utilisent tous le protocole TCP-IP

EXEMPLE UN INTRANET SIMPLE



IDENTIFICATION DES CLIENTS : ADRESSE MAC

Chaque carte Ethernet possède un code d'identification unique sur 48 bits : adresse MAC inscrit en dur sur la carte.
La forme de notation courante est : 00-0E-35-F8-9A-38. Les deux premiers octets identifient le constructeur.

PROTOCOLE IP : TRAMES IP

- Chaque trame comporte un CRC généré par la carte réseau. Si le CRC est correct IP transmet le paquet sinon il le supprime sans en informer l'expéditeur.
- IP n'est pas conçu pour fournir une transmission garantissant l'intégrité des données sont rôle principal est le routage.
- IP fonctionne un peut comme un émetteur de radio diffusion : il se contente de fournir de l'information à un destinataire que celui-ci soit présent ou non.

PROTOCOLE TCP-IP

TCP assure la transmission ordonnée des paquets. Ses principales fonctions sont les suivantes :

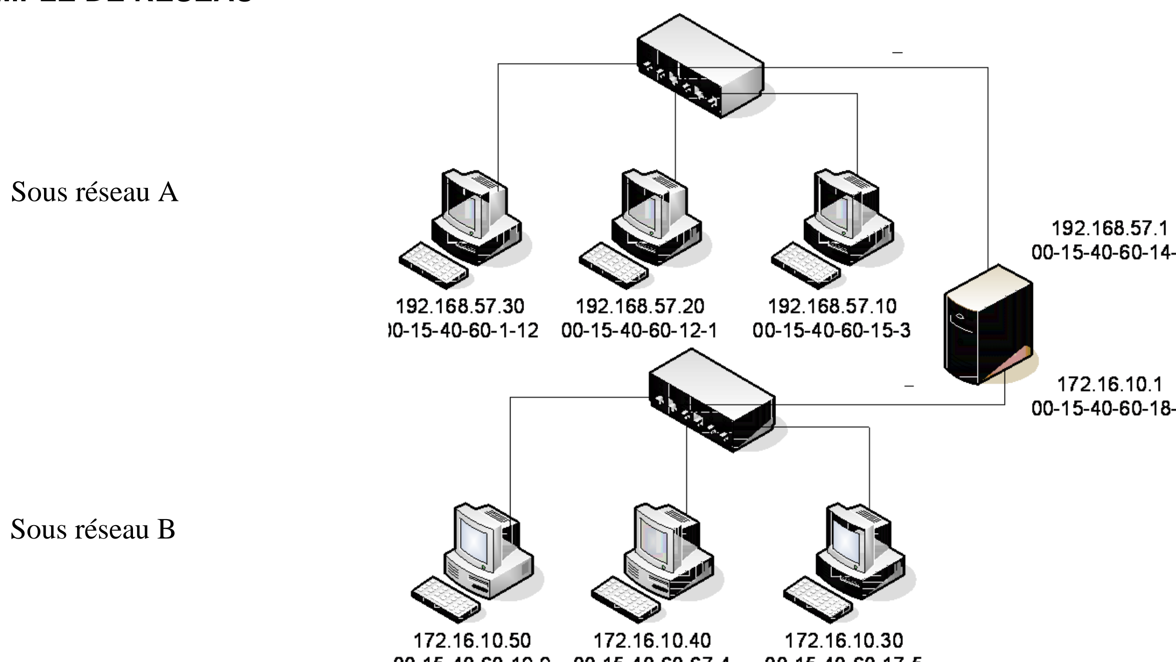
- Acquittement des paquets (accusé de réception des paquets)
- Séquencement des paquets :
  - Comme les paquets peuvent emprunter différents chemin, certains paquets peuvent être retardés donc arriver dans un ordre différents. TCP remet les paquets dans l'ordre.
- Contrôle du flux :
  - les paquets sont conservés dans un tampon et sont transmis au destinataire quand les paquets sont complets.
- Gestion des erreurs :
  - TCP garanti que les paquets reçus sont intègre en redemandant la retransmission des blocs erronés.

ADRESSES IP (V4)

Les adresses IP sont sur 32 bits et sont notées sous la forme de 4 octets décimaux
Exemple :

```
11000000 10101000 00001010 0000101000
   192  .  168       10   .    40
```

EXEMPLE DE RESEAU



CLASSES DE RESEAU

Classe du réseau	Plage d'adresses X : attribuées par L'INA (NIC), ---- attribuées localement				Rem : valeur des bits 23..31
Classe A	0XXX XXXX	---- ----	---- ----	---- ----	0..126
Classe B	10XX XXXX	XXXX XXXX	---- ----	---- ----	128..191
Classe C	110X XXXX	XXXX XXXX	XXXX XXXX	---- ----	192.223
Rebouclage	0111 1111	---- ----	---- ----	---- ----	127
Multidiffusion	1110 XXXX	---- ----	---- ----	---- ----	224..239
Réservé	1110 XXXX	---- ----	---- ----	---- ----	240..255

Adresses non routables (RFC 1918)

Pour se prémunir du manque d'adresses IP, l'idée est de mettre de coté des plages d'adresses que les utilisateurs peuvent exploiter sans demander l'autorisation à L'INA. Si ces adresses son utilisées directement sur internet, elles ne posent aucun conflit puisqu'elles sont ignorée par les routeurs. Ces adresses sont utilisées dans les réseaux Intranet.

Adresses non routables
10.0.0.0 .. 10.255.255.255
172.16.0.0 .. 172.31.255.255
192.168.0.0 .. 192.168.255.255

MASQUES DE SOUS-RESEAU

Les masques de sous-réseau ont pour but de segmenter les réseaux en sous réseau.
Les bits de sous-réseau sont composés des bits de poids fort. Le bits de masquage vaut 1.

Exemple dans le sous réseau de classe C, les adresses sont de la forme : 199.34.57.XX
et ont donc pour masque : 255.255.255.0

```
11000111 00100010 00111001 XXXXXXXX
11111111 11111111 11111111 00000000
```

ADRESSES RESERVEES

- L'adresse itinéraire par défaut : 0.X.Y.Z. Comme cette adresse fait parti de la classe A toutes les adresses 0.X.Y.Z doivent être réservées (16 millions d'adresses gaspillées).
- Adresse de bouclage 127.0.0.1
  Les messages transmis à cette adresse restent à l'intérieur du composant logiciel IP (16 millions d'adresses gaspillées).
- Adresse de diffusion. Permet de diffuser la même message à tous les ordinateurs d'un sous réseau. Elles est formée avec les bits du sous réseau à 1.
  Exemple : sous réseau de classe C : 199.34.57.0 → adresse de diffusion 199.34.57.255
  Adresse du routeur par défaut. Chaque sous-réseau doit comporter au moins un routeur pour pouvoir communiquer avec les autres réseaux. Par convention elle est la première adresse du sous réseau. Avec l'exemple précédent elle serait 199.34.57.1

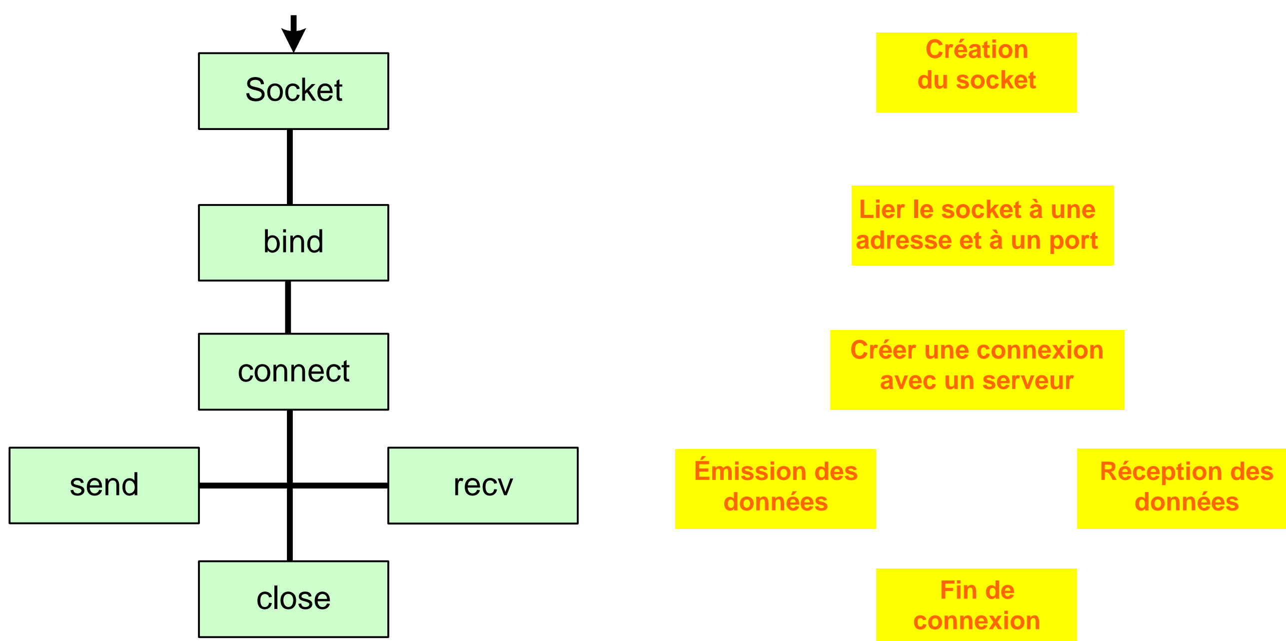
PROGRAMMATION RESEAU

- Les requêtes réseau sur internet utilisent exclusivement le protocole IP
  - Ce protocole est mis en œuvre à travers l'API Socket définie par l'université de Berkeley.
  - Cet API relativement complexe est devenue un standard de fait dans la programmation réseau en particulier dans la programmation réseau internet.

SOCKETS ET PORTS

Lors d'une communication réseau, on est amené a utiliser différents protocoles correspondant chacun à un programme différents. Par exemple au sein d'une même machine on peut avoir un serveur Web, un serveur de courriel électronique, un serveur de transfert de fichier. Comme tout ces programmes utilisant TCP-IP, il faut une information supplémentaire pour distinguer ces applications : cette information est le numéro de port codé sur 16 bits.
Un socket est une adresse IP associée avec un numéro de port. Par exemple un serveur Web utilise le numéro de port 80, un serveur Smtp (serveur de courriel) utilise le port 25, un serveur FTP utilise les ports 20 et 21…

UTILISATION DES SOCKETS PAR LE CLIENT



Utilisation d'un socket par un client

CONNEXION A UN SERVEUR

Exemple : utilisation du programme telnet.
La commande suivante permet de se connecter au service date d'un serveur situé au NIST (National Institute of Standards and Technology) à Boulder dans le Colorado. Ce service fourni l'heure d'une horloge atomique au césium.

```
telnet o time-A.timefreq.bldrdoc.gov 13
```

- 13 représente le port auquel est attaché le service date,
- `time-A.timefreq.bldrdoc.gov` est l'adresse du service qui doit être converti en adresse IP (132.163.4.102) par un serveur de nom situé sur le réseau Internet.

La commande (page suivante) fourni comme résultat 28/09/03 08:54:03

```
Telnet time-A.timefreq.bldrdoc.gov
Microsoft Telnet> o
( à ) time-A.timefreq.bldr.gov 13
Connexion à time-A.timefreq.bldr.gov ...
52910 03-09-28 08:54:03 50 0 0 307.7 UTC(NIST) *
Perte de la connexion à l'hôte.
```

EXEMPLE APPEL DU SERVICE DATE AVEC JAVA

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

public class SocketTest
{
  public static void main(String[] args)
  {
    try {
      Socket s = new Socket("time-A.timefreq.bldrdoc.gov", 13);
      BufferedReader in = new BufferedReader
                          (new InputStreamReader(s.getInputStream()));
      String line;
      while (true)
      {
        line = in.readLine();
        if (line == null) break;
        System.out.println(line);
      }
    }
    catch (IOException e)
    {
      System.out.println(e);
    }
  }
}
```

Ouverture du socket

Lecture de la donnée fournie par le serveur

```
Console [<arrêté> C:\Program Files\Java\j2re1.4.1_01\bin\javaw.exe (28/09/
52910 03-09-28 09:10:26 50 0 0   0.0 UTC(NIST)  *
```

LES ADRESSES INTERNET

Une adresse Internet est composée de 4 octets (16 pour IPv6). Cette adresse n'est pas parlante.

- La classe `InetAddress` permet de traduire un nom d'ordinateur en adresse Internet et réciproquement :
- La méthode statique `getByName` retourne un objet `InetAdddress` qui encapsule la séquence de 4 octets :

  `InetAddress adr = InetAddress(getByName("Time-A.time.bldrdoc.gov");`

  // retourne un objet encapsulant l'adresse 132.163.4.102

  `Byte[] octets = adr.getBytes();`
  // retourne un tableau contenant les 4 octets précédents.

- Certains serveurs au trafic très important ont plusieurs adresses : getAllByName permet de connaître toutes ces adresses :
  - `InetAddress[] adr = InetAddress(geAllByName(host));`
- Un ordinateur a parfois besoin de connaître sa propre adresse. Si on se contente de demander l'adresse de localhost la réponse est systématiquement 127.0.0.1 ! Il faut utiliser getLocalHost pour obtenir son adresse :
  - `InetAddress adr = InetAddress(getLocalHost());`

L'exemple page suivante affiche l'adresse Internet de l'ordinateur hôte, ou toute autre adresse Internet d'un autre ordinateur spécifié sur la ligne de commande.

```
 1 import java.net.*;
 2 public class InetAddressTest
 3 {
 4    public static void main(String[] args)
 5    {
 6       try {
 7          if (args.length > 0)
 8          {
 9             String host = args[0];
10             InetAddress[] addresses
11                = InetAddress.getAllByName(host);
12             for (int i = 0; i < addresses.length; i++)
13                System.out.println(addresses[i]);
14          }
15          else
16          {
17             InetAddress localHostAddress
18                = InetAddress.getLocalHost();
19             System.out.println(localHostAddress);
20          }
21       }
22       catch (Exception e) { e.printStackTrace(); }
23    }
24 }
```

```
<terminated> 34-InetAddressTest [Java Application] C:\j2sdk1
jeanbart.serveftp.com/80.170.147.210
```

QUELQUES EXEMPLES D'APPLICATIONS RESEAU

Lire le contenu d'une URL

- L'URL de classe représente un localisateur de ressources uniformes
- Une ressource peut être un dossier, un annuaire, ou une référence à un objet plus complexe tel qu'une requête à une base de données
- Lire le contenu d'une URL permet de récupérer le contenu d'une page HTML, d'un script CGI, ...
- Le port à utiliser est contenu implicitement dans le nom du protocole exemple (http: => port 80) mais peut être explicitement donné dans l'URL :
  Exemple : `http://www.exemple:8080`

Classe URL

Constructeur

`public URL(String spec) throwsMalformedURLException`

Créé un objet URL à partir de sa représentation sous forme de String.

Paramètre:

- `spec` – la chaîne pour parser l'URL.

Exception :

- `MalformedURLException` si l'URL spécifie un protocole inconnu.

Méthodes principales de la classe URL

- **`String getFile()`**
  Chemin et fichier précisé dans l'URL
- **`String getHost()`**
  Nom du domaine
- **`String getProtocol()`**
  Protocole utilisé dans l'URL.
- **`InputStream openStream()`**
  Ouvre une connexion vers cette URL et retourne un objet InputStream pour lire à partir de cette connexion

Exemples

Pour l'URL :
`http://www.eduscol.education.fr/D0102/liste-mots-alphabetique.txt`

Avec le code suivant

```
System.out.println(webURL.getProtocol());
System.out.println(webURL.getHost());
System.out.println(webURL.getFile());
System.out.println(webURL.getDefaultPort());
```

On obtient :

```
http
www.eduscol.education.fr
/D0102/liste-mots-alphabetique.txt
80
```

Lecture d'une page HTML

```
URL webURL = null;
try
{
  webURL = new URL("http://www.google.fr");
  BufferedReader is = new BufferedReader(new InputStreamReader(webURL
      .openStream()));
  String line;
  while ((line = is.readLine()) != null)
  {
    System.out.println(line);
  }
  is.close();
}
catch (MalformedURLException e)
{
  System.err.println("Load failed: " + e);
}
catch (IOException e)
{
  System.err.println("IOException: " + e);
}
```

Connexion à un serveur SMTP

- Un serveur SMTP est un serveur de courrier électronique fonctionnant suivant le principe suivant :
  - Un client communique des données au serveur SMTP en précisant un destinataire.
  - Le serveur mémorise les données jusqu'à ce que le client destinataire vienne récupérer ces données.
- Les commandes envoyées au serveur se font sous la forme de chaines ASCII.

Commandes principales d'un serveur SMTP

- Pour envoyer un courrier électronique il faut ouvrir une connexion socket sur le port 25 également appelé port SMTP (Simple Mail Transfert Protocol).

```
Socket s = new Socket("nom_du_serveur_smtp", 25);
PrintWriter out = new PrintWriter(s.getOutputStream());
in = new BufferedReader(new InputStreamReader(s.getInputStream()));
```

- Une fois connecté au serveur, il faut envoyer des entêtes spécifiant les commandes SMTP :

  HELO ordinateur envoyant le courrier<**cr,lf**>
  MAIL FROM: <adresse e-mail de l'expéditeur><**cr,lf**>
  RCPT TO: <adresse e-mail cible><**cr,lf**>
  DATA<**cr,lf**>
  Subject: sujet <**cr,lf**>
  From: <adresse e-mail de l'expéditeur><**cr,lf**>
  To: <adresse e-mail cible><**cr,lf,cr,lf** >
  Corps du message (n'importe quel nombre de lignes) <**cr,lf.cr,lf**>
  QUIT

Codes d'erreur dans SMPT

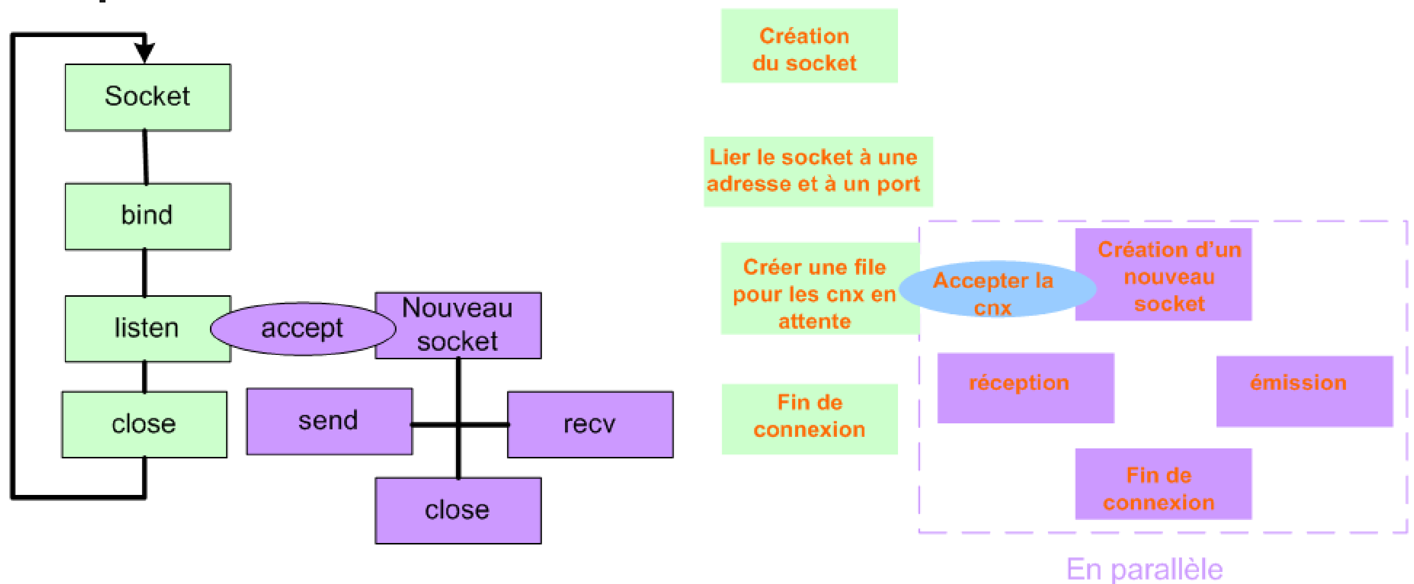
Code	Description
250	Pas d'erreur
421	*Domain* service not available, closing transmission channel. Service non disponible, canal en fermeture
432	A password transition is needed.
450	Requested mail action not taken: mailbox unavailable. Action non effectuée : boîte-aux-lettres non disponible [Ex., bloquée par un autre utilisateur]
451	Requested action aborted: local error in processing. Action arrêtée : erreur de traitement
452	Requested action not taken: insufficient system storage. Action non effectuée : manque de ressources système.
454	TLS not available due to temporary reason. Encryption required for requested authentication mechanism.
458	Unable to queue messages for node *node*.
459	Node *node* not allowed: *reason*.
500	Command not recognized: *command*. Syntax error. Erreur de syntaxe, commande non reconnue [y compris des erreurs de type "ligne de commande trop longue"]
501	Syntax error, no parameters allowed. Erreur de syntaxe dans des paramètres ou arguments
502	Command not implemented. Commande non implémentée
502	Command not implemented. Commande non implémentée
503	Bad sequence of commands. Mauvaise séquence de commandes
504	Command parameter not implemented. Paramètre de commande non implémenté
521	*Machine* does not accept mail.
530	Must issue a STARTTLS command first. Encryption required for requested authentication mechanism.
534	Authentication mechanism is too weak.
538	Encryption required for requested authentication mechanism.
550	Requested action not taken: mailbox unavailable. Action non effectuée : boîte-aux-lettres non disponible [Ex., boîte-aux-lettres non trouvée, pas d'accès]
551	User not local; please try *forwardpath*. Utilisateur non local ; essayer (sans relais automatique)
552	Requested mail action aborted: exceeded storage allocation. Action arrêtée : manque de ressources de stockage
553	Requested action not taken: mailbox name not allowed. Action non effectuée : nom de boîte-aux-lettres non autorisé (par exemple, une erreur de syntaxe dans le nom de la boîte]
554	Transaction failed. Transaction échouée.

Utilisation d'un serveur SMTP: pour l'envoi d'un courriel

```
try
{
  s = new Socket("smtp.mamadoo.fr", 25);
  out = new PrintWriter(s.getOutputStream());
  in = new BufferedReader(new InputStreamReader(s.getInputStream()));
  String hostName = InetAddress.getLocalHost().getHostName();
  System.out.println(hostName); receive();
  send("HELO " + hostName); receive();
  send("MAIL FROM: <" + "dupond@free.fr" + ">"); receive();
  System.out.println("->MAIL FROM");
  send("RCPT TO: <" + "lajoie@free.fr" + ">"); receive();
  send("DATA"); receive();
  send("Subject: test mail");
  send("From: dupond@free.fr"); send("To: lajoie@free.fr\r\n");
  send("corps du message");
  send(".");   receive();
  send("END"); receive();
  s.close();
}
catch (Exception e)
{
  System.out.println(e);
}
```

```
// Affiche le résultat des commande mail
public void receive() throws IOException
{
   String line = in.readLine();
   if (line != null)
     System.out.println(line);
}

public void send(String s) throws IOException
{
   out.print(s);
   out.print("\r\n");
   out.flush();
}
```

UTILISATION DES SOCKETS PAR UN SERVEUR



Utilisation d'un socket par un serveur

- Coté serveur : classe `Java.net.ServerSocket` :
  - Creation du socket serveur
    `ServerSocket` crée un objet socket de serveur qui examine un port :
    `ServerSocket(int port) throws IOException;`
  - Attente d'une connexion.
    C'est une méthode bloquante (synchrone). Cette méthode retourne un objet socket avec lequel le serveur peut communiquer avec le client
    `Socket accept() throws IOException;`
  - Fermer la connexion socket du serveur
    `void close() throws IOException;`

IMPLEMENTATION D'UN SERVEUR

Exemple : réalisation d'un serveur qui retourne en écho les requêtes à une seul client.

- Notre serveur utilise le port 8189 (non utilisé en standard)
- La classe ServerSocket est utilisée pour établir la connexion
- La commande

```
ServerSocket s = new ServerSocket(8189);
```

Demande au serveur d'attendre indéfiniment jusqu'à ce qu'un client se connecte.

- Une fois le client connecté on peut se servir de cet objet pour lire et écrire sur le réseau :

```
BufferedReader BfrIn = new BufferedReader
                      (new InputStreamReader(in.getInputStream()));
PrintWriter BfrOut = new PrintWriter(in.getOutputStream(), true);
```

```
 1 import java.io.*;
 2 import java.net.*;
 3 public class EchoServer
 4 {
 5    public static void main(String[] args ) {
 6       try {
 7          ServerSocket s = new ServerSocket(8189);
 8          Socket in = s.accept( );
 9          BufferedReader BfrIn = new BufferedReader
10             (new InputStreamReader(in.getInputStream()));
11          PrintWriter out = new PrintWriter
12             (in.getOutputStream(), true /* autoFlush */);
13          out.println( "Frapper BYE pour quitter." );
14          while (true) {
15             String line = BfrIn.readLine();
16             if (line == null) break;
17             else {
18                out.println("Echo: " + line);
19                if (line.trim().equals("BYE")) break;
20             }
21          }
22          in.close();
23       }
24       catch (Exception e) {
25          e.printStackTrace();
26       }
27    }
28 }
```

```
Telnet 127.0.0.1
Microsoft Telnet> o
( à ) 127.0.0.1 8189
Connexion à 127.0.0.1...
Frapper BYE pour quitter.
-------------------------
Echo: -------------------------
abcdef
Echo: abcdef
BYE
abcdef
Echo: abcdef
BYE
Perte de la connexion à l'hôte.

Appuyez sur une touche pour continuer...
```

Test du serveur

Lancer telnet en utilisant les paramètres suivants :

- Serveur 127.0.0.1 (adresse de retour locale)
- Port 8189

- N'importe qui dans le monde peut maintenant utiliser ce serveur, il suffit qu'il connaisse son adresse IP et le port qu'il utilise.
- Ce serveur possède cependant un inconvénient majeur : il ne peut servir qu'un seul client à la fois. Si plusieurs clients se connectent le premier client connecté monopolise le serveur. Ce défaut sera corrigé plus loin.

SERVEUR MULTITACHES (MULTITHREAD)

Typiquement un serveur un serveur est exécuté en permanence sur un ordinateur dédié à cette tâche.
Plusieurs clients situés sur le réseau Internet doivent pouvoir se connecter simultanément.
Pour pouvoir servir un client le serveur doit être capable de créer une tâche dédiée à ce client, tout en attendant qu'un autre client se connecte. Un serveur doit donc avoir des capacités multitâches.

- Chaque fois qu'un client se connecte, on crée une nouvelle tâche (thread) chargé de la connexion entre le serveur et le client.
- Le programme principal reboucle en attendant en attendant la prochaine connexion.

```
for (;;)
{
    Socket in = s.accept( );
    Thread t = new EchoHandlerMultitache(in);
    t.start();
}
```

- La classe EchoHandlerMultitache dérive de la classe Thread et contient la boucle de communication avec le client.

```
class EchoHandlerMultitache extends Thread
{
  private Socket in;
  public EchoHandlerMultitache(Socket i)  { in = i;  }
  public void run()  // méthode lancée par start
  {
      try {
         BufferedReader BfrIn = new BufferedReader
                        (new InputStreamReader(in.getInputStream()));
         // ...
         while (true) {
           //...
         }
         in.close();
      }
      catch (Exception e)  { System.out.println(e); }
}
```

TIMEOUT DES SOCKETS

- Si l'ordinateur distant (serveur) serveur ne répond pas, l'application sera bloquée.
- Il est possible de fixer une butée temporelle à la transaction à l'aide de `setTimeout(int millisecondes)`

```
Socket s = new Socket(...);
s.setTimeout(10000); // ferme le socket après 10 sec
```

- Lorsque la butée temporelle est atteinte toute opération de lecture déclenche une exception `InterruptIOException` interceptée de façon suivante:

```
try
{
     while ((String  line = in.readLine())!= null)
     {
        // traitement de la ligne
     }
}
catch (InterruptIOException e) { /*gestion du timeout*/ }
```

- Le constructeur peut également se bloquer si la connexion n'est pas établie:
  `Socket(String host, int port);`
- Il est possible de résoudre ce problème en construisant d'abord un socket non connecté, puis en le connectant avec un timeout :

```
Socket s = new Socket();
s.connect(new InetSocketAdress(host, port), timeout);
```