Programmation Java

Version.1.10 du 11/05/98

Michel Buffa, Peter Sander, Richard Grin,
Jean-Philippe Orsini

Université de Nice

Email : `buffa@unice.fr, sander@essi.fr, grin@unice.fr,`

`orsini@essi.fr`

Remerciements

- **Je tiens à remercier les personnes suivantes qui ont grandement contribué à la mise à jour de ce cours**
  - Jacques Farré (jf@essi.fr)
  - Jacques Chazarain (jmch@unice.fr)
  - Richard Grin (grin@unice.fr)
  - Peter Sander (sander@essi.fr)
  - Jean-Philippe Orsini (orsini@essi.fr)

Introduction

Introduction à Java

Généralités

- **Java est un véritable langage de programmation !**
  - Syntaxe proche de C++, orienté objet, d'une utilisation simple, implémentant un maximum de sécurité, il permet d'exécuter des programmes au travers d'une machine virtuelle.
- **Il est fourni avec un ensemble de "packages" : les classes de base du JDK (Java Developper's Toolkit)**
  - Sans ces classes de base, Java n'est rien. Elles fournissent les fonctionnalités de base (entrées/sorties, réseau, etc...). Le langage lui-même se trouve dans le package (ensemble de classes) **`java.lang`** !
- **Java et l'internet**
  - Du fait de son fonctionnement au travers d'une machine virtuelle, Java est un langage "qui voyage".

Introduction à Java
Caractéristiques de base

- **Java est un langage**

1. simple,
2. orienté objet,
3. distribué,
4. interprété,
5. robuste,
6. sûr,
7. indépendant de l'architecture (portable),
8. mobile,
9. performant (pas pour toutes les applications, mais ça va venir...)
10. multithreadé,
11. dynamique,
12. exécuté non pas au travers d'un interpréteur, mais au travers de l'émulation d'une machine virtuelle, ou d'un puce JavaOS
13. possédant une riche API : le JDK
14. véritable explosion depuis son lancement : nombreux produits commerciaux : IDEs, APIs spécialisées, etc...

Introduction à Java

Simplicité (apparente ?)

- **Basé sur C/C++, mais...**
    - pas de pointeurs,
    - pas de structures ni de macros (**`struct, typedef et #defines`**),
    - pas de gestion de la mémoire laissée au programmeur (**`malloc()`** et **`free()`** sont morts, paix à leurs âmes...)

Java offre toute la fonctionnalité d'un langage de programmation puissant, mais débarrassé des caractéristiques prêtant à confusion.

Sa syntaxe est proche de celle de C/C++ mais (fort heureusement !) a été adaptée pour en permettre une approche plus rapide et surtout en gommer certains aspects critiqués ou provocant de nombreuses erreurs de programmation.

L'expérience prouve que Java est un langage simple à prendre en main, mais en aucun cas limité. La révision 1.1 propose même des technologies objets très avancées comme les RMIs (Remote Method Invocations).

En revanche, comme tous les langages objets, il faut une certaine expérience avant de tirer parti au mieux de ses possibilités, et surtout de son API très riche.

Introduction à Java

Simplicité (apparente ?)

- **N'a-t-on rien perdu par rapport au C/C++ ?**

  Non, les suppressions évoqués précédemment réduisent de moitié les erreurs les plus fréquentes du C et du C++.

  En outre, la suppression des macros, si elle peut sembler une hérésie, augmente de beaucoup la lisibilité des programmes.

  L'héritage multiple au sens courant a disparu, mais existe sous une forme différente grâce au concept d'"interfaces".

- **Java est-t-il un beau langage ?**

  La notion de "beau" est subjective, mais oui, Java est un langage extrêmement agréable à programmer et permet dans certains cas de développer des applications beaucoup plus rapidement qu'avec d'autres langages.

- **Java est peu gourmand en mémoire**

  - La machine virtuelle n'a besoin que de 215k pour s'exécuter.
  - Java dans une montre à quartz ? Oui, c'est possible !

Introduction à Java

Orienté objet, distribué

- **Orienté objet**

  Java reprend la technologie objet du C++ et y ajoute quelques améliorations. Il y enlève aussi un gros morceau (l'héritage multiple, la généricité et d'autres encore...) mais sans vraiment y perdre. Nous y reviendrons...

  Contrairement à C++, Java est complètement objet, ne demeurent que quelques types primitifs (int, char) hérités du C possèdant des “wrapper classes”.!

- **Distribué au travers des classes du JDK**

  Java lui-même n'est pas distribué. En revanche, il est fourni avec un ensemble de classes qui intègrent une gestion plus ou moins transparente du réseau.

  Par exemple, il est possible, en quelques lignes de code, de récupérer un fichier au travers du protocole HTTP ou FTP, de programmer un serveur de socket pouvant accepter des connexions en parallèle, etc...

  Ce genre de programmation est aujourd'hui un des gros attraits de Java.

Introduction à Java

Robuste

- **Nombreuses vérifications à la compilation et à l'exécution**

  Lors de la phase de compilation, Java s'assure qu'il n'y a aucun problème, que toutes les erreurs pouvant empêcher l'exécution du programme ont bien été traitées (ouverture d'un fichier inexistant ou accès mémoire en dehors des bornes d'un tableau par exemple).

  Cette vérification permet d'éviter qu'un problème grave ne corrompe la machine virtuelle ou le système qui l'exécute. N'oubliez pas que les programmes Java "voyagent" d'une machine à l'autre ! Vous ne voudriez pas qu'un programme arrive dans votre navigateur Web et plante votre machine !

- **Java fournit au travers de la machine virtuelle et du langage**

  - vérification de pointeur nul,
  - vérification des bornes des tableaux,
  - traitement très puissant des exceptions,
  - vérification du byte code (pas de virus, très difficile de "cracker" des programmes compilés, etc...).
  - hmmm... mais désassemblage assez facile avec la release 1.0 du JDK !

Introduction à Java

Indépendant de l'architecture

- **Indépendance au niveau des sources (JDK) et du Byte Code (machine virtuelle)**

  Le compilateur Java du JDK ne fournit pas directement du code machine exécutable, il produit un code portable, du byte-code, interprétable par la machine virtuelle.

  Ce byte-code est indépendant de l'architecture. Chaque machine possédant une machine virtuelle peut l'exécuter. Actuellement Java peut être exécuté sous presque tous les Unix du marché, sous Windows 3.1/95/NT, sous MacOS, dans des Net Computers (NC) ou des systèmes embarqués possédant une puce JavaOS (téléphones portables, télévisions, etc...).

- **Compilation directe possible (traduction C++ puis compilation aujourd'hui), mais adieu la transportabilité du code !**

- **Performances quand même excellentes depuis l'arrivée des JITs (Just In Time compilers...)**
  - Compilation lors de la phase d'interprétation, compilation “au vol”.
  - On approche la vitesse du C/C++ optimisé, mais on y est pas encore.

Introduction à Java

Sûr : le vérificateur de byte-code

- **Contrôle de tout code chargé avant instanciation pour exécution**

  Le code Java est testé plusieurs fois avant d'être exécuté sur votre machine.

  Le code passe par un vérificateur de byte-code. Celui-ci teste le format des fragments de code et applique un algorithme qui détermine leur légalité. Un code légal ne modifie pas les pointeurs, ne viole pas les droits d'accès aux objets et ne tente pas d'altérer les objets ou les classes.

  Une application indépendante peut se permettre plus de choses qu'une application s'exécutant dans un navigateur Web (applet). Le class loader et le security manager sont différents dans les deux cas.

- **Vérificateur de byte-code**

  - Le code ne cause aucun dépassement de capacité positif ou négatif de la pile opérande,
  - les types de paramètres de tous les codes opération sont corrects,
  - aucune conversion illégale de données (entiers en pointeurs, etc...),
  - les accès aux objets sont légaux (publics, privés, protégés... nous y reviendrons !)

Introduction à Java

Sûr : le Class Loader

- **Les classes sont chargées dans un NameSpace indépendant et identifié en fonction de l'origine du code chargé**

  Les classes sont, comme nous le verrons plus tard, le code de base servant à la construction par instanciation des objets utilisés par une application.

  Elles peuvent être chargées par le réseau (applets et classes développées par l'utilisateur) ou encore locales, liées à la machine virtuelle en cours d'utilisation.

  Le **`Class Loader`** participe donc à la sécurisation de Java en séparant les espaces-noms pour les classes locales et les classes provenant du réseau. Cela limite les possibilités de voir un virus ou une application “cheval de troie” envahir votre machine sans prévenir!

  Lorsqu'on a besoin d'une classe donnée, la recherche est effectuée en priorité parmi les classes locales. De cette manière, aucune classe importée ne peut se faire passer pour une classe système, même si elle en usurpe le nom.

  Quoique...

Introduction à Java

Le code Java est mobile

- **Java intègre, au travers de son JDK, une interface commune permettant une compatibilité source totale entre les architectures**

  De par son indépendance vis-à-vis de l'architecture, Java est par essence mobile. Il intègre cependant d'autres standards de mobilité afin de faciliter le développement d'applications portables.

  Par exemple, la longueur des types de base est toujours identique quelle que soit l'architecture utilisée, les codes des caractères sont partout les mêmes, le codage des mots de 16 ou 32 bits est le même (adieux big et little endian !), etc...

  De même, la librairie graphique, l'AWT (Abstract Window Toolkit) permet de développer des interfaces graphiques sans se préoccuper du système d'exploitation qui sera utilisé.

Introduction à Java

Java est interprété

- **La compilation produit du byte-code non lié à une plate-forme**
- **Celui-ci est interprété par une machine virtuelle**
- **Pas de phase d'édition de liens avec des librairies existantes, c'est un garant de la portabilité du langage.**
- **Excellentes performances depuis peu**
  - Attention, le JDK proposé gratuitement par Sun est le plus lent du marché! Tout est écrit en Java (compilateur, machine virtuelle, etc...).
  - Des produits infiniment plus performants, mais commerciaux sont disponibles chez différents développeurs.
  - Par exemple : compilation de 2 millions de lignes/seconde sur Pentium 133 avec Symantec Café.
  - L'interpréteur est remplacé par un Just In Time Compiler (Solaris, nombreuses versions sous Windows 95/NT par Symantec, Microsoft, Borland, Netscape). 300.000 appels de méthodes/procédures par seconde sur un Sun Sparc 10 avec le JIT de Sun. Presque identique à du C/C++ natif équivalent.

Introduction à Java

Java est multithreadé

- **Un Thread est une partie de code, un “flot d’instructions” s’exécutant en concurrence avec d’autres Threads dans un même processus**

  L’utilisation du multithreading permet à un seul programme d’effectuer plusieurs activités simultanément.

  Les threads, parfois appelés “processus légers” permettent de développer des programmes multitâches de manière plus simple et performante qu’avec la programmation système classique (fork() et exec()...), dont le code est par ailleurs difficilement portable.

  Les Threads faisant partie intégrante du langage Java (jusque dans certains de ses mots-clé), ils sont plus faciles à utiliser que leurs équivalents C/C++.

  Les avantages principaux du multithread sont des performances plus élevées en matière d’interactivité et un meilleur comportement en temps réel, bien que ce dernier soit en général dépendant du système.

  Java intègre les Threads de manière transparente (garbage collector, horloge, chargement des images ou des sons), mais permet également au programmeur de développer ses propres Threads de façon très simple.

  Des outils de gestion de ressources critiques (sémaphores, moniteurs temps réels) et de synchronisation sont également disponibles, et leur utilisation est d’une grande simplicité.

Introduction à Java

Java est multithreadé

- **Sous Java les Threads**
  - ne sont pas toujours natifs,
  - sont intégrés par la machine virtuelle : garbage collector, horloge...,
  - sont mis à la disposition de l'utilisateur au travers de classes et d'interfaces du JDK : Thread, Frame, Runnable, etc...

Introduction à Java

Java est dynamique

- **Un programme est constitué de plusieurs classes**
- **Lorsqu'une classe inconnue dans le programme est requise, on la recherche :**

  1. localement (dans le JDK local au navigateur web ou à la machine virtuelle) et ensuite si elle n'est pas trouvée...
  2. à l'endroit d'où le programme a été chargé

  Java tire au maximum parti de la technologie objet, la machine virtuelle n'essaie pas de connecter tous les modules composant une application avant l'exécution (pas d'éditions de liens).

  Les classes de définition des objets sont chargées simultanément mais indépendamment, en fonction de leur besoin dans l'application. Si une application nécessite une classe non présente sur la machine qui l'exécute, Java ira la chercher à l'endroit d'où l'application provient.

Applications indépendante et applets

Objectifs de ce chapitre

- **Connaître les différences entre applet et application indépendante**
- **Savoir compiler du code Java et l'exécuter**
- **Apprendre à configurer un environnement de développement Java**

Applications indépendante et applets

application indépendante

- **Comme n'importe quel autre langage, un code Java dispose pour son exécution d'un point d'entrée.**

- **A la différence d'autres langages ce point d'entrée possède deux composantes :**

1. La machine virtuelle
2. Une méthode :

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

... c'est le cas des applications indépendantes. C'est en codant cette méthode que l'on a la possibilité d'indiquer le point de départ de toute application Java. C'est le “programme principal” pour reprendre une ancienne terminologie plus tellement adaptée...

3. Remarque : en fait chaque classe peut avoir une méthode **`main()`**, mais celle qui sera exécutée sera celle de la classe passée en paramètre à la machine virtuelle (la classe maître). Très utile pour debugger les classes indépendamment : le main contient une série de tests de validation de la classe.

Applications indépendante et applets
Applets

- **Les applets ne sont pas des applications indépendantes, elles ne possèdent pas forcément de méthode `main()`.**

- **Les applets doivent être exécutées au travers d'un browser (Netscape, Internet Explorer...).**

- **Il y a plusieurs points d'entrée d'exécution dans une applet, et la méthode `main()` n'en fait pas partie !**

- **En réalité, le browser contient en interne :**

  1. la machine virtuelle,
  2. sa propre méthode

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

  Une applet est une application Java très particulière car elle s'exécute dans un browser. Ce dernier doit donc pouvoir comprendre Java mais aussi effectuer un certain nombre de contrôles sur le code qu'il va exécuter. Ce dernier point justifie (entre autre) le besoin de ne pas mettre en contact direct la machine virtuelle avec un code qui la contrôlerait. Le Browser fait donc tourner une application indépendante qui se charge d'exécuter des applets après avoir fait les vérifications nécessaires.

Applications indépendante et applets
Applets

- **Mais alors, où commence l'exécution de mon applet ?**

  Il a été défini une liste de méthodes qui sont utilisées comme différents points d'entrées, en fonction du cycle de vie de l'applet : **`init(), start(), stop(), destroy()`**...

- **Pourquoi plusieurs points d'entrée ? Qu'est-ce que le cycle de vie d'une Applet ?**

  Une Applet est un programme qui s'exécute dans un browser, son utilisation dépend du bon vouloir de l'utilisateur du browser. Ce dernier peut revenir sur une page et masquer momentanément la page contenant l'applet. Dans ce cas, il faut arrêter les threads qui tournent, stopper les animations, le son, etc... Si l'utilisateur revient dans la page, il faut relancer les threads arrêtés, d'où les méthodes **`stop()`** et **`start()`**.

  Nous verrons le détail de ces méthodes et du cycle de vie des applets dans le chapitre qui leur est consacré.

Applications indépendante et applets Résumé

- **Une application indépendante Java**
  - ...possède un point d'entrée unique, la méthode **`main()`**, de prototype :

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

  - ...est exécutée directement par la machine virtuelle Java.

- **Une applet Java**
  - ...est exécutée par un browser,
  - ...ne dispose pas de point d'entrée **`main()`**,
  - ... dispose de points d'entrée appelés sous le contrôle de l'application indépendante embarquée par le browser.

Créer une application Java
Les bases à connaître

- **L'entité de base de tout code Java est la classe**
- **Tout fichier source doit contenir une seule classe publique**
- **Tout fichier source doit porter le nom de la classe publique qu'il contient, suivi du suffixe .java**
- **Une classe peut contenir uniquement des données (attributs) et/ou des méthodes**
- **Une méthode peut contenir uniquement des données et/ou du code.**

```
public class HelloWorld {
    // déclaration de données globales
    int donneeGlobale;

    // méthodes
    public static void main(String argv[]) {
        int donnee_locale;
        System.out.println("Hello world");
    }
}
```

Fichier HelloWorld.java

Créer une application Java

Les bases à connaître

En Java, tout se trouve dans une classe. Il ne peut y avoir de déclarations ou de code en dehors du corps d'une classe.

La classe elle-même ne contient pas directement du code. Une classe contient des déclarations de variables globales, que l'on appelle des "attributs", et des méthodes (équivalents des fonctions).

Les "attributs" sont visibles par toutes les méthodes de la classe qui peuvent les utiliser. Ces attributs seront "possédés" par chaque objet instance de la classe.

Le code se trouve exclusivement dans le corps des méthodes, mais ces dernières peuvent aussi contenir des déclarations de variables locales (visibles uniquement dans le corps de la méthode).

Pour que votre code soit réutilisable, pour augmenter sa lisibilité et sa localisation et pour faciliter la compilation et le chargement "au vol" des classes, il ne peut y avoir qu'une seule classe publique par fichier source.

En effet, une application étant composée d'un ensemble de classes, nous avons vu que ces dernières sont chargées par la machine virtuelle lorsqu'on en a besoin. Pour pouvoir retrouver rapidement les classes à charger, il est indispensable de retrouver le nom du fichier à partir du nom de la classe.

L'environnement de développement Java
Variables d'environnement

- **Trois variables d'environnement sont très importantes**

    1. CLASSPATH : chemin de recherche des classes de base.
    2. JAVA_HOME : répertoire de base de l'installation du JDK
    3. PATH : doit contenir le répertoire contenant le compilateur java (javac), la machine virtuelle (java), le debugger (jdb), etc... En général $JAVA_HOME/bin

- **Ces variables sont souvent positionnées automatiquement lors de l'installation d'une IDE (symantec café, etc...), qui inclue le JDK.**

- **Sous Unix, les positionner !**

- **Les browsers incluant leur propre machine virtuelle et leurs propres classes repositionnent le CLASSPATH. Si l'installation est bien réalisée, ceci est fait de manière transparente.**

L'environnement de développement Java

Compilation

- **Compilation à l'aide du JDK. Attention, avec une IDE, se référer à la documentation spécifique, la procédure est certainement différente.**
  - Dans une fenêtre de commande (shell Unix ou fenêtre MS/DOS) :

```
javac HelloWorld.java
```

Si la compilation s'est bien passée, on devrait obtenir un fichier :

```
HelloWorld.class
```

- **Sinon... erreurs de compilation !**

Le compilateur indique précisément la ligne à laquelle l'erreur s'est produite. Le message est en général explicite :

```
HelloWorld.java:9 Method printl(java.lang.String) not found
in class java.io.PrintStream.
System.out.printl("Hello World");
```

Nous avons ici les informations suivantes :

1. Erreur à la ligne 9,
2. L'erreur a eu lieu dans l'appel de **`System.out.printl(...)`**
3. Visiblement, la méthode **`printl()`** n'existe pas. En effet la bonne méthode s'appelle **`println()`**.

L’environnement de développement Java
Compilation

- **Exécution du code avec le JDK (IDEs voir la documentation spécifique)**
  - Dans une fenêtre de commande (shell Unix ou fenêtre MS/DOS) :

```
java HelloWorld
```

- **Attention au CLASSPATH si l’erreur est du style :**

```
Can’t find class HelloWorld
```

ou bien...

```
Can’t find class HelloWorld.class
```

Cette erreur signifie que la machine virtuelle n’a pas pu trouver la classe **`HelloWorld.class`** dont elle avait besoin pour exécuter l’application.

L’erreur classique est que la variable CLASSPATH n’inclut pas le répertoire courant dans le chemin de recherche des classes.

Une autre possibilité est que vous avez essayé d’exécuter un code qui n’a pu être compilé correctement (aucun fichier .class généré)

L’environnement de développement Java

Premier exercice

- **Créez l’application HelloWorld !**

1. Editez le fichier source **`HelloWorld.java`** à l’aide d’un éditeur de texte,
2. Créez une classe publique **`HelloWorld`** contenant une méthode **`main()`** affichant le texte “Hello world!”,
3. Compilez le fichier source. En cas de problème vérifiez que votre environnement est bien initialisé,
4. Exécutez l’application,
5. Modifiez légèrement l’application en introduisant par exemple des erreurs, et recommencez les étapes 1 à 4,
6. Bon, il est temps d’apprendre les différents éléments de base du langage Java, sa syntaxe, etc... de manière à développer des application plus évoluées !

Eléments traditionnels du langage
Objectifs de ce chapitre

- **Connaître l'architecture élémentaire d'un programme Java**
- **Assimiler la structure lexicale du langage**
- **Savoir utiliser les types offerts par Java**
- **Lister les instructions du langage**

Eléments traditionnels du langage
Introduction

- **Java est orienté objet**

  Java a toutes les caractéristiques d'un langage objet; il permet de définir des classes comportant des descriptions d'attributs (variables) et d'opérations (méthodes) utilisables sur des objets.

  Les objets communiquent entre eux et réagissent aux événements issus de leur environnement par le biais de messages.

  Le langage est enrichi par un ensemble de classes fournies en standard avec le JDK. Elles fournissent en particulier les fonctionnalités réseau, temps réel, entrées/sorties, mathématiques, etc... Ainsi que les éléments d'interface graphique.

- **Java est fortement typé**

  Toute donnée manipulée par Java doit être typée. Le type peut être soit un type de base (int, float, char, boolean...) soit un nom de classe du JDK ou créée par l'utilisateur.

- **Java est déclaratif**

  Tout objet ou variable doit être déclaré avant d'être manipulé. La déclaration comporte de manière classique un type et l'identificateur de donnée.

```
int nb = 10;
```

Eléments traditionnels du langage
Introduction

- **Java est structuré**
  - Structuré en packages (ensembles de classes), classes et méthodes.
  - Lors de l’exécution, les instructions de contrôle de flot (branchements, boucles...) sont elles mêmes structurées (emboîtement possible).

```
for (i=0; i < 10; i++) {
   if (i == 5) {
      System.out.println(“milieu!”);
   }
}
```

- **Java est... beau !**

  Après avoir programmé en C, en C++ ou en bien d’autres langages, passer à Java n’est pas très difficile. En revanche le contraire est souvent pénible car Java est très agréable et permet de développer vite et simplement des applications complexes.

Eléments traditionnels du langage
Architecture d'un programme Java

- **Programme Java = ensemble de classes pouvant être regroupées en packages**

  Java est un langage orienté objet. Le composant de base d'un programme Java est donc la classe, qui agit comme modèle permettant de "fabriquer" des objets. Les classes peuvent être regroupées en modules appelés packages. Les fonctionnalités spécifiquement objet de Java seront étudiées dans un autre chapitre.

- **Classe = attributs + méthodes**

  Un classe comporte un ensemble de définitions d'attributs et un ensemble de méthodes que l'on pourra utiliser sur tous les objets de cette classe.

- **Méthode = unique unité de code**

  Le code source d'un programme Java doit se situer uniquement à l'intérieur du corps des méthodes (à l'exception de certains codes statiques, comme nous le verrons plus tard).

- **Une classe publique par fichier. Le fichier porte le nom de la classe suivi du suffixe ".java"**

  Chaque classe publique, c'est-à-dire destinée à pouvoir être utilisée par les objets d'une autre classe quelconque, doit être définie dans un fichier .java qui porte son nom.

Eléments traditionnels du langage

Architecture d'un programme Java

- **Exemple : fichier Point.java**

```
public class Point {
   // Les attributs

   private int x;
   private int y;

   // Les méthodes

   public void setXY(int px, int py) {
      x = px;
      y = py;
   }

   public int getX() {
      return x;
   }

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

Eléments traditionnels du langage

Architecture d'un programme Java

- **Rappels :**
  - Une application indépendante Java possède un point d'entrée : la méthode **`main()`**

```
public class MonApplication {

   ...
   public static void main(String argv[]) {

      // code de départ de l'application

      ...

   }
}
```

  - La méthode **`main()`** d'une applet java est incluse dans le browser, les applets ne possèdent pas de point d'entrée **`main()`**.

Eléments traditionnels du langage
Structures lexicales

- **Jeu de caractère ASCII classique**

  Le jeu de caractères utilisé lors de l'écriture d'un source Java est le code ASCII habituel.

- **Identificateurs**

```
int variable;
int autreVariable = 10;
double variableReelle;
short petitEntier;
float le_$prix;
```

  Un identificateur désigne tout objet que le programmeur souhaite manipuler. Il peut s'agir d'une classe, d'un objet, d'une variable ou d'une méthode.

  Un identificateur Java commence par une lettre et peut ensuite contenir d'autres lettres, des chiffres, un caractère souligné (_), ou encore un dollar ($)

- **Terminateur d'instructions**

```
int x = 1;
int y = 10;
int somme;

somme = x + y;
```

  Il s'agit du caractère point-virgule, déjà utilisé par le C, le C++, Modula 2, etc...

Eléments traditionnels du langage

Structures lexicales

- **Commentaires d'une seule ligne**

```
// Ceci est un commentaire d'une ligne
int x = 10; // centre du cercle
```

- **Commentaires sur plusieurs lignes**

```
/* Ceci est un long commentaire qui
   s'étale sur plusieurs lignes... */
```

Les commentaires s'emploient de la même manière qu'en C++, soit avec // en début de ligne, soit en encadrant le commentaire entre /* et */

Les commentaires peuvent être placés à n'importe quel endroit du source.

- **Documentation automatique par javadoc**

```
/** Cette méthode permet de faire le
* ...traitement suivant ... blah blah..
*...
*/
```

Javadoc est un utilitaire très pratique permettant de générer automatiquement une documentation au format HTML à partir des commentaires définis comme ci-dessus.

Il existe de nombreux mots-clé que l'on peut insérer dans le commentaire, pour spécifier l'auteur, le rôle de la méthode, etc... on peut insérer du HTML...

Eléments traditionnels du langage

Types élémentaires

Table 1: Types élémentaires de Java

Nom	Taille en bits	Exemples de littéraux
byte	8	1
short	16	2
int	32	2, -12, 300
long	64	2L
float	32	3.14, 3.1E12, 2e12
double	64	0.5d
boolean	1	true ou false
char	16	‘a’, ‘\t’ (tab), ‘\u0000’ (NULL en unicode)

Les types Java sont indépendants du compilateur et de la plate-forme d’exécution. Ils sont intégrés aux spécifications du langage, ce qui garantit une parfaite portabilité du byte-code.

Les valeurs numériques entières sont représentées pas byte, short, int et long, les réelles par float et double.

Un boolean n’est pas considéré comme une valeur numérique. Ses seules valeurs possibles sont true et false.

Les caractères ne sont plus représentés sur 8 bits mais sur 16 par l’intermédiaire du code unicode. Ceci permet de représenter potentiellement tous les alphabets de la planète, en particulier nos alphabets accentués.

Eléments traditionnels du langage
Chaînes de caractères

- **En C**
  - Calcul des longueurs de chaînes,
  - gestion de la fin de chaîne (caractère de code ASCII zéro),
  - pas de vérification de débordements.
- **En C++**
  - Utilisation de bibliothèques pas toujours portables ni standards (bien qu'il existe un simili standard).
- **En Java**
  - **`String`** et **`StringBuffer`** sont deux classes fournies avec le JDK permettant un traitement très riche des chaînes de caractères.

```
String nom = "toto";
```

Pas de marqueur de fin de chaîne, pas de calcul de la longueur lors de la création, vérification des débordements, puissants mécanismes de conversion des types.

`String` permet de représenter des chaînes constantes, **`StringBuffer`** de taille variables.

Etudiées en détail dans la suite du cours.

Eléments traditionnels du langage

Variables et attributs

- **Déclaration**

```
int x;
float f;
char a;
```

- **Déclaration avec initialisation**

```
int x = 123; // initialisation
float f = 0.0;
String toto = “toto”;
```

Tous les objets doivent être déclarés avant d’être utilisés. Syntaxe :

```
<type> <identificateur> [ = expression];
```

L’expression optionnelle permet d’affecter une valeur lors de l’initialisation.

Table 1: Valeurs par défaut des types de base, ***pour les attributs d’une classe***

Type	Valeur par défaut
byte	0
short	0
int	0
long	0L
float	0.0f
double	0.0d
char	‘\u0000’
boolean	false
référence objet (pointeur)	null

Eléments traditionnels du langage

Variables

- **Initialisation : contrôle du compilateur**

```
int i;
while(i < 10) {
   System.out.println("i vaut " + i);
   i++;
}
```

La compilation produit le message d'erreur suivant

```
test.java:6 Variable i may not have
been initialized.
        while(i < 10) {
1 error
```

Le compilateur javac effectue de nombreuses vérifications quant à la qualité du code qu'il compile.

Il est en particulier capable d'avertir le programmeur d'une incohérence éventuelle.

Ici, une donnée n'est pas initialisée avant son utilisation. Cette donnée est *une variable locale à une méthode*. Dans ce cas, elle n'est pas initialisée avec des valeurs par défaut.

Le seul cas d'initialisation automatique avec une valeur par défaut est celui des *attributs d'une classe (ou variable de classe)*.

Dans l'exemple, il aurait fallu mettre :

```
int i = 0;
```

Eléments traditionnels du langage
Variables

- **Transmission des variables par valeur**
  - Tous les attributs et variables possédant un type de base (int, char...) sont transmis par valeur.
  - Dans ce cas, un paramètre de méthode contient directement la valeur qu'on lui assigne. On effectue une copie de la valeur de la variable transmise. On ne modifie donc pas l'original.

```
int valeur = 10;
Point p = new Point();
p.move(valeur); // valeur non
                // modifiable
```

- **Transmission des variables par référence**
  - Toutes les instances d'objets Java sont transmises par référence (par adresse).
  - Dans ce cas, la variable représente seulement le moyen d'accéder à l'objet et non pas l'objet directement. Elle contient donc l'adresse mémoire de l'objet. La particularité d'une référence par rapport à un pointeur est de ne permettre d'accéder qu'à la donnée qu'elle pointe.
  - Ce mode de gestion permet un partage aisé d'objets au sein d'une application ainsi que l'assurance d'avoir des performances optimales lors du passage de gros objets en paramètres.

Eléments traditionnels du langage
Variables

- **Transmission des variables par référence (suite)**
  - Java ne permet pas de choisir le mode de passage en paramètre des objets, comme en C++ par exemple. Mais en imposant une solution unique simple et efficace, les concepteurs de Java prétendent garantir l'homogénéité de la programmation et faciliter la maintenance d'une application. Ca se discute...
  - Pour résumer, en Java on distingue deux sortes d'objets
    1. Les objets primitifs (int, char, etc...)
    2. Les objets référencés (tous les autres, attention, ceux de la classe String en font partie !)

- **Possibilité de forcer les paramètres en constantes : les rendre non modifiables**

```
public class myClass {
   public methode1(final int x) {
      int y = x + 2;            Autorisé
      x = x + 3;                Interdit !
   }
}
```

Eléments traditionnels du langage

Variables

- **Blocs et visibilité : structuration**

```
// début du premier bloc
{
   int i = 10;
   ...
   // début du second bloc
   {
      int i = 0;
      int j = 20;
      ...
   }
   // Fin du second bloc
   // j n'est plus visible
}
// fin du premier bloc
```

Produit à la compilation

```
bloc.java: 7: Variable 'i' is already defined in this method
                    int i = 0;
1 error
```

- **Java n'autorise pas la redéfinition au sein d'un bloc d'une variable déjà visible. Le compilateur produit donc une erreur.**
  - Amélioration de la lisibilité et diminution des erreurs. Pas d'ambiguïté lors de l'utilisation d'une variable.

Eléments traditionnels du langage
Tableaux

- **En Java les tableaux sont des objets référencés. Le type tableau est assimilable à une classe**
  - Ils gèrent leur taille, on peut y accéder au travers de l'attribut length

```
int[] t = new int[10];
...
int longueur = t.length;
// longueur vaut maintenant 10
```

  - Contrôle dynamique des débordements. Un dépassement de capacité produit une exception.

- **Déclaration d'un tableau**

```
int tableau1[];
int[] tableau2;

int matrice[][];

int[] x, y[]; // équivalent à :

int x[], y[][];
```

  - En java il n'est pas possible de définir des tableaux de taille statique à la déclaration.
  - Un tableau doit être alloué dynamiquement, comme tout objet Java (à l'aide de l'opérateur new)

```
int tab[10]; // ne compile pas
```

Eléments traditionnels du langage
Tableaux

- **Point controversé**

```
int tab[10]; // ne compile pas
```

- **Pourquoi ne peut-on pas avoir ?**

```
int tab[10];
```

  - Ce serait bien que ça compile, non ?

- **Ou bien**

```
int tab[] = new int(10);
```

- **Les tableaux et la présence de types primitifs tels que int, char etc... est un gros point de controverse.**

- **Rappel : on ne peut pas faire**

```
int x = new int(3);
```

- **Mais on peut faire**

```
int x = Integer.parseInt(“12”);
```

Eléments traditionnels du langage
Tableaux

- **Allocation et initialisation**

```
int tab1[] = new int[10];
int mat[][] = new int[4][5];

int tab2[] = {1,2,3,4}
```

- Un tableau n'est pas un type élémentaire. Une variable de type tableau est donc une référence sur un tableau. pour créer ce tableau nous utilisons l'opérateur new. C'est une allocation dynamique.
- il faut fournir la taille de la première des dimensions d'un tableau alloué par new.

```
int tab[][] = new int[4][];
```

- Il est également possible de créer le contenu d'un tableau lors de la déclaration en fournissant la liste complète de ses éléments.
- Depuis le JDK1.1, on peut créer le contenu d'un tableau lors de toute affectation (tableaux anonymes)

```
String voitures = new String[]
                {"Renault", "peugeot"};
```


Eléments traditionnels du langage
Tableaux

- **Utilisation**

```
int tab1[] = new int[10];
...
// utilisations correctes
tab1[0] = 100;
tab1[1] = 200;

// mais ceci n'est pas correct
tab1[10] = 1000;
```

La compilation produit le message suivant :

```
java.lang.ArrayIndexOutOfBoundsException: 10 at Tab-
Test.main(TabTest.java:6)
```

- En Java les tableaux sont toujours indexés par des int et le premier élément commence à l'indice 0.
- tab[tab.length - 1] est le dernier élément du tableau
- Si le tableau est alloué avec une taille inférieure à zéro ou si on tente d'accéder un élément ne se trouvant pas entre le premier et le dernier élément compris, une exception est générée.
- N.B : tab.length est manipulé comme une variable

- **Les chaînes de caractères ne peuvent être manipulées comme des tableaux de caractères (oubliez le C et le C++ !)**

Eléments traditionnels du langage
Expressions

- **Opérandes : représentés par des variables ou des littéraux**

```
int i = 2;
int resultat = 3 * i;
```

- **Opérateurs**
  - Java impose une parfaite identité de types des opérandes d’un opérateur binaire. Contrairement à C et à C++, il n’y a *presque* aucune conversion implicite de type.

```
int x, y=6;
float z=3.14f;
...
x = z/y;
```

...génère le message suivant à la compilation :

```
TstCast.java:7: Incompatible type for =. Explicit
cast needed to convert float to int.

          x = z/y;

1 error
```

Il faudrait écrire :

```
x = (int)z /y; // division entière
```

ou :

```
x = (int) (z /(float) y); // division
                          // réelle
```

Eléments traditionnels du langage
Expressions

- Les casts implicites apparaissent quelquefois dans les expressions numériques, lorsqu'il n'y a aucun risque de perte de précision

```
int i = 2;
double d = 4;
...
d = i;
```

Ces exemples marchent

```
double d1 = 8;
int i = 2;
double d2 = d1/i;
```

- Jamais de cast implicite si perte de précision !

Eléments traditionnels du langage
Ordre d'évaluation des opérateurs

Table 1: Ordre d'évaluation

<table>
<tr><th>Associativité</th><th></th></tr>
<tr><td>Droite-Gauche</td><td>. {} ()</td></tr>
<tr><td>Droite-Gauche</td><td>++ --</td></tr>
<tr><td>Droite-Gauche</td><td>! ~ (conversion de type)</td></tr>
<tr><td>Gauche-Droite</td><td>* / %</td></tr>
<tr><td>Gauche-Droite</td><td>+ -</td></tr>
<tr><td>Gauche-Droite</td><td><< >> >>></td></tr>
<tr><td>Gauche-Droite</td><td>< > <= >= instanceof</td></tr>
<tr><td>Gauche-Droite</td><td>== !=</td></tr>
<tr><td>Gauche-Droite</td><td>&</td></tr>
<tr><td>Gauche-Droite</td><td>^</td></tr>
<tr><td>Gauche-Droite</td><td>|</td></tr>
<tr><td>Gauche-Droite</td><td>&&</td></tr>
<tr><td>Gauche-Droite</td><td>||</td></tr>
<tr><td>Droite-Gauche</td><td>? :</td></tr>
<tr><td>Droite-Gauche</td><td>= *= /= %= += -= <<=
>>= >>>= &= ^= |=</td></tr>
</table>

Eléments traditionnels du langage

Ordre d'évaluation des opérateurs

- **Remarque importante relative au tableau précédent :**

```
...
if ((x != 0) && (y/x == 2)) {
   ....
```

- **... ne produit pas d'erreur si `x` vaut zéro**

Eléments traditionnels du langage
if...else

- **Le branchement conditionnel en Java s'effectue exclusivement à partir d'une expression booléenne.**

```
if(expression_booléenne){
   instructions
} else {
   instructions
}
```

  - Le bloc `else` est optionnel
  - Les accolades `{}` sont obligatoires dès que l'on désire exécuter plus d'une instruction dans le bloc du `if` ou du `else`.
  - Exemples :

```
if(i < 0) i = 0; // avec int i;
if(error) {
   fin = true;
   i = 0;
} else
   i++;
```

```
int i=10;
if(i) i=0; // Non! i n'est pas booléen
```

Eléments traditionnels du langage
expression conditionnelle

- **Autre formulation “héritée” du C (pas forcément une bonne chose !)**

```
int x := (expression_booléenne ? si vrai : si faux)
```

Eléments traditionnels du langage
switch

- **Débranchement multiple**

```
switch(expression){
   case expr1:
      instructions;
      [break;]

...
   case exprN:
      instructions;
      [break;]
   default:
      instructions;
      [break;]
}
```

- L'expression évaluée dans le **`switch`** doit retourner un type dont les valeurs peuvent être énumérées **`(int, boolean, char)`**.
- Il est possible de mettre plusieurs instructions dans un **`case`**.
- Si on ne met pas de **`break`** à la fin des instructions d'un **`case`**, les instructions du **`case`** suivant sont exécutées. Voir le cas du **`case 1`** de l'exemple page suivante.

Eléments traditionnels du langage *switch*

- **Un exemple**

```
switch(count % 3) {

   case 0:
      methode1(count);
      break;

   case 1:
      count = -1;

   case 2:
      methode2(count);

} // fin du switch
```

  - Si `(count % 3)` vaut 1, les instructions suivantes seront exécutées :

```
count = -1;
methode2(count);
```

Eléments traditionnels du langage
boucle for

- **Syntaxe**

```
for(init; test; incrément){
   instructions;
}
```

- **Exemple**

```
int i;

for(i =0; i < 10; i++) {
   System.out.println(i);
   ...
   if(i == 1) continue;

   if(val_i_ok(i)) break;
}
```

- L'instruction **`break`** permet de sortir de la boucle et de continuer l'exécution après le bloc d'instructions du **`for`**.
- L'instruction **`continue`** force l'exécution à l'itération suivante. Dans l'exemple, lorsque i vaut 1, la ligne suivante **`if(val_i_ok...)`** n'est pas exécutée, on reprend sur la ligne **`System.out...`** avec **`i = 2`**

Eléments traditionnels du langage
boucle for

- **Déclaration de variables dans la boucle for**
  - Il est possible de déclarer une ou plusieurs variables dans la partie init de la boucle **`for`**. Dans ce cas la visibilité de ces variables est limitée au bloc d'instructions lié à la boucle.

```
for(int i = 0; i < 10; i++) {
   instructions;
}
```

`Visibilité de i`

- **Etiquettes**
  - Les instructions **`break`** et **`continue`** peuvent être exécutées avec des étiquettes afin de permettre des ruptures de séquences sur plusieurs niveaux d'emboîtement de boucles. A EVITER !

```
fori: for(int i=0; i < 10; i++){
         ...
         for(int j=0; j < 4; j++){
            ...
            if(...)
               continue fori;
            if(...)
               break fori;
         }
      }
```

Eléments traditionnels du langage

while et do...while

- **Syntaxe**

```
while(expression_booléenne){
   instructions;
}
```

```
do {
   instructions;
} while(expression_booléenne);
```

- **Exemple**

```
int i = 10;

while(i > 10) {
   ...
   i--;
}
```

```
int j = 0;

do {
   ...
   j++;
} while(j < 100);
```

- **Il est possible d'utiliser des étiquettes pour faire des ruptures de séquence avec plusieurs niveaux d'emboîtements. La syntaxe est la même que celle employée page précédente avec la boucle `for`.**

Concepts objets de Java

Objectifs de ce chapitre

1. Connaître les concepts de la programmation par objet et les mettre en oeuvre avec Java
2. Créer une classe avec des attributs et des méthodes
3. Instancier des objets et leur envoyer des messages
4. Définir un graphe d'héritage
5. Créer une classe abstraite
6. Définir et utiliser des interfaces
7. Regrouper des classes en packages

Concepts objets de Java

Définition d'un objet

- **Qu'est-ce qu'un objet ?**
  - Un ensemble de données (attributs) et d'actions (méthodes) formant un tout indépendant.
  - Les attributs décrivent l'état de l'objet.
  - Les méthodes permettent de faire "vivre" l'objet (en lui faisant effectuer des actions et peut-être changer son état).
  - C'est... "l'instanciation d'une classe" (pages suivantes)
- **Exemple d'objet : un bouton poussoir dans une interface graphique**
  - Attributs : une image, un label, une couleur de fond, une police de caractères, etc...
  - Méthodes : se dessiner, réagir quand on clique dessus.
- **Autre exemple : un pacman**
  - Attributs : couleur, vitesse, direction du déplacement, état (mort, vivant), etc...
  - Méthodes : se dessiner, avancer, manger les pastilles,...

Concepts objets de Java
Définition d'un objet

- **Un objet peut être**
  - une valeur (instance) d'un type abstrait, une "représentation" d'une classe.
  - une abstraction d'un objet physique du monde réel, c'est-à-dire dont on a retenu uniquement les caractéristiques intéressantes lors de la modélisation.

- **Un objet est caractérisé par**
  - Un ensemble **d'attributs** typés et nommés représentant ses propriétés statiques. Ces attributs portent des valeurs attachées à l'objet. L'ensemble de ces valeurs à un instant donné constitue l'état de l'objet à cet instant,
  - Un ensemble de **méthodes** définissant son comportement et ses réactions aux stimulations externes. Ces méthodes implémentent les algorithmes invocables sur cet objet,
  - Une **identité** qui permet de le distinguer des autres objets, de manière unique.

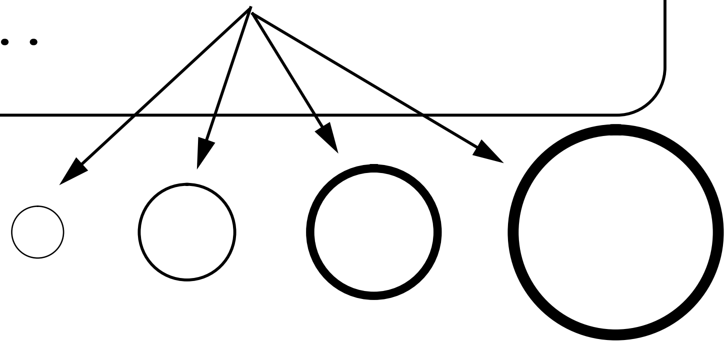
Concepts objets de Java

Définition d'une classe

- **Qu'est-ce qu'une classe ?**
  - C'est un modèle de définition pour les objets
    - en termes d'attributs,
    - en termes de méthodes.
  - Les objets sont des représentations dynamiques (instanciation), "vivantes", du modèle défini pour eux au travers de la classe.

- **Exemple : la classe Cercle**

```
public class Cercle {
   int rayon;
   int epaisseur_trait;
   ... // autres attributs
   ...
}
```



Instanciation, même attributs mais de valeurs différentes

Concepts objets de Java

Définition d'une classe

- **Une classe définit les modèles des objets qui en seront instanciés**
  - dans la classe sont définis des attributs et des méthodes qui seront présents **dans chaque objet** qui sera instancié de la classe considérée. Dans l'exemple de la page précédente, on définit une classe Cercle, contenant entre autres un attribut rayon et un attribut epaisseur_trait. **Chaque** cercle instancié de cette classe possédera **son** propre rayon et **sa** propre épaisseur du trait.
- **Les méthodes définies dans la classe existent de même pour chaque objet et ont un comportement identique dans chaque objet tout en y étant localisé**
  - Supposons que la classe Cercle fournisse la méthode changeRayon(int new_rayon), permettant de modifier la valeur de l'attribut rayon. Si un des cercles fait appel à cette méthode, il changera **son propre rayon, pas celui des autres** cercles dont il ignore probablement l'existence.

```
int rayon = 50;
Cercle c = new Cercle(rayon);
c.dessineToi();
c.changeRayon(rayon+10);
c.dessineToi();
```

Concepts objets de Java

Définition d'une classe

- **Instanciation. Mais que cela signifie-t-il exactement ?**
  - la création d'un objet à partir d'une classe est appelé **instanciation**. Cet objet est **une instance** de sa classe.
  - Une instanciation se décompose en trois phases

1. Création et initialisation des attributs en mémoire, à l'image d'une structure.
2. Appel de méthodes particulières, les constructeurs, définies dans la classe, dont l'étude est faite un peu plus loin...
3. Renvoi d'une référence sur l'objet (son identité) maintenant créé et initialisé.

Concepts objets de Java

Encapsulation

- **L'encapsulation est la possibilité de ne montrer de l'objet que ce qui est nécessaire à son utilisation, les avantages en sont :**
  - Simplification de l'utilisation des objets,
  - Meilleure robustesse du programme,
  - Simplification de la maintenance globale de l'application.

```
public class Cercle {
   private int cx, cy;
   int rayon;
   int epaisseur_trait;
   ... // autres attributs
   ...
   void move(int dx, int dy) {
      cx +=dx;
      cy +=dy;
   }
}
```

= Visible de l'extérieur

= privé, visible uniquement par les méthodes de la classe où ils sont déclarés

Concepts objets de Java

Encapsulation

- **Gestion de la visibilité des objets**
    - L'encapsulation permet d'offrir aux utilisateurs d'une classe la liste des méthodes et éventuellement des attributs utilisables depuis l'extérieur.
    - Cette liste de services exportables est appelé **l'interface** de la classe et elle est composée d'un ensemble de méthodes et d'attributs dits **publics.**
    - Les méthodes et attributs réservés à l'implémentation des comportements internes à l'objet sont dits **privés**. Leur utilisation est exclusivement réservée aux méthodes définies dans la classe courante.

Concepts objets de Java

Encapsulation : avantages

1. **Simplification de l'utilisation des objets.**

   Pour utiliser un objet, inutile de connaître son implémentation, la connaissance de son interface suffit. Par exemple, il est inutile de savoir que les coordonnées du centre du cercle sont stockées dans deux variables **`cx`** et **`cy`**, qui sont de type **`int`**.

2. **Meilleure robustesse du programme**

   Une fois un objet testé et développé, un “bug” ne peut pas être généré par du code extérieur à l'objet.

   Par exemple, si on tente par erreur de mettre une valeur de type **`int`** dans la variable **`cx`** ou **`cy`**, ce ne sera pas permis.

   Nous somme donc obligé d'appeler la méthode publique **`move(int dx, int dy)`**. Si un bug apparaît, c'est alors dans l'objet qu'il faut le chercher (notre méthode publique est-elle capable de traiter ce type d'erreur ?)

3. **Simplification et maintenance de l'application**

   En fonction de ce que nous avons dit précédemment, ce point en est une conséquence logique.

Concepts objets de Java

Héritage et polymorphisme

- **L'héritage permet de créer des classes par combinaison de classes déjà existantes**

```
Class ObjetGraphique {
    int x, y;
- - - - - - - - - - - - - - -
    void setXY(int, int);
    Point getXY();
    void Dessiner()
}
```

Héritage

```
Class Cercle extends ObjetGraphique{
    float rayon;
- - - - - - - - - - - - - - - - - - -
    void Dessiner();
}
```

Partie héritée de ObjetGraphique

```
    int x, y;
- - - - - - - - - - - - - - -
    void setXY(int, int);
    Point getXY();
    void Dessiner()
```

Partie héritée de ObjetGraphique

```
Class Rect extends ObjetGraphique{
    int width, height;
- - - - - - - - - - - - - - - - - - -
    void Dessiner();
}
```

```
    int x, y;
- - - - - - - - - - - - - - -
    void setXY(int, int);
    Point getXY();
    void Dessiner()
```

```
Rect r1 = new Rect();
```

Ce dont dispose l'objet r1, instance de Rect

```
    int x, y;
    int width, height;
- - - - - - - - - - - - - - -
    void setXY(int, int);
    Point getXY();
    void Dessiner()
```

Concepts objets de Java

Héritage et polymorphisme

- **Définition de l’héritage**
  - Lorsqu’une classe hérite d’une autre classe, elle bénéficie automatiquement de ses définitions d’attributs et de méthodes. Elle peut y ajouter ses propres définitions, ce qui permet facilement de constituer un ensemble fonctionnellement plus riche.
  - Les liens d’héritage spécifient une **hiérarchie** de classe qui peut être constituée par une démarche de **spécialisation**, ou au contraire de **généralisation**.

- **Définition du polymorphisme**
  - Lorsqu’un objet est instancié d’une classe, il peut être vu comme instance de n’importe laquelle des classes héritées par la classe dont il est instancié.
  - En fait l’instance d’un tel objet **est toutes les classes à la fois**. Le rectangle `r1` est à la fois un objet de la classe Rect **et** un objet de la classe `ObjetGraphique`.
  - Le polymorphisme permet donc d’écrire :

```
ObjectGraphique r = new Rect();
```

Concepts objets de Java

Héritage et polymorphisme, surcharge et redéfinition

- **Définition de la redéfinition (redéfinir une méthode héritée)**
  - Lorsqu'un attribut ou une méthode ont été définis dans une classe et sont redéfinis dans une classe dérivée (qui en hérite), les éléments visibles sont ceux redéfinis dans la classe dérivée. Les éléments de la classe de base (héritée) sont alors masqués.

```
Class Rect extends ObjetGraphique {

   void move(int dx, int dy);
      ...
}
Class Cercle extends ObjetGraphique{

   void move(int dx, int dy);
      ...
}
```

  - On peut avoir les mêmes méthodes `move()` dans des classes héritant les unes des autres. Dans ce cas, c'est la classe la plus dérivée de l'objet qui détermine la méthode à exécuter, sans que le programmeur ait à faire des tests sur le type de l'objet à traiter.

Concepts objets de Java

Héritage et polymorphisme, surcharge, redéfinition

- **Définition de la surcharge**
  - La surcharge est la capacité des objets d'une même hiérarchie de classes à répondre différemment à une méthode de même nom, **mais de paramètres différents**. Par exemple, la classe **`Rect`** peut avoir plusieurs méthodes **`move()`** acceptant des paramètres différents en nombre et/ou en types. En fonction du type et du nombre de paramètres lors de l'appel, la méthode correspondante sera choisie.

```
Class Rect extends ObjetGraphique {

   void move(int dx, int dy);
   void move(int vitesse);
   void move(boolean direction);
   ...
}
```

Concepts objets de Java
Modularité

- **Découpage en sous-systèmes**
  - Ne pas lier les classes les unes avec les autres.
  - Utiliser la généricité des abstractions.
- **Réutilisabilité et évolutivité**
  - Documenter et homogénéiser l'interface publique des classes d'un même module.
- **Un système est dit modulaire s'il est découpé en sous-systèmes possédant les caractéristiques suivantes**
  - Forte cohérence interne
    - Vos classes doivent être validées indépendamment du code qui les exécute. Tous les éléments (méthodes et attributs) qui en font partie et qui peuvent remettre la cohérence de son fonctionnement en cause doivent être isolés de l'extérieur et se trouver dans la partie privée de la classe.
    - Les méthodes publiques de l'interface de votre classe doivent permettre l'accès sous contrôle à ces éléments lorsque c'est nécessaire. Ces méthodes seront alors le seul point d'entrée dans votre classe pour le monde extérieur.
  - Faible couplage avec les autres sous-systèmes
    - Voir page suivante...

Concepts objets de Java

Modularité

- Faible couplage avec les autres sous-systèmes
  - Utilisez la généricité issue des abstractions. Cela évite qu'une classe soit liée à une autre classe, mais permet de la lier à un modèle de classe (classe abstraite ou interface, voir plus loin...)
  - Exemple : supposons une classe **`FeuilleDePapier`**. Si nous décidons que cette classe ne sait gérer que des **`Rectangles`** ou des **`Cercles`**, le jour où nous créerons une classe **`Triangle`**, nous devrons modifier la classe **`FeuilleDePapier`** si nous voulons qu'elle gère aussi les instances de **`Triangle`**.
  - Si au contraire, nous définissions que **`FeuilleDePapier`** gère uniquement des **`ObjetsGraphiques`**, alors aucune modification ne sera requise si **`Triangle`** hérite de **`ObjetGraphique`**, ce qui est logiquement le cas.
- la modularité permet de réutiliser des modèles d'analyse et de conception ainsi que du code et favorise l'évolutivité

Concepts objets de Java

Utilisation des classes, visibilité

- **Exemple des ObjetGraphique, sans héritage**

```
public class Cercle {
   // attributs privés à notre classe
   private int x, y;
   private float rayon;

   // interface publique

   // Les constructeurs
   public Cercle(int cx, int cy,
                          float r) {
   x = cx;
   y = cy;
   rayon = r;
}

   public Cercle() {
      x = 10;
      y = 10;
      rayon = 10f;
   }

   public void Dessine() {
      System.out.println("ici je
                   dessine un cercle");
   }

   public void modifieRayon(float r) {
      rayon = r;
   }
}
```

Concepts objets de Java

Utilisation des classes, visibilité

- **Attributs**

```
[visibilité] type nom [= expression] ;
```

- **`visibilité`** peut avoir l'une des valeurs suivantes :

1. **`private`** : l'attribut est inaccessible à l'extérieur de la classe,
2. **`public`** : l'attribut est manipulable à l'extérieur,
3. les autres valeurs seront étudié avec l'héritage...

- **`expression`** permet de fournir une valeur par défaut à l'attribut lors de la création de l'objet.

- **Méthodes**

```
[visibilité] [type retour] nom (liste paramètres);
```

- **`visibilité`** a le même comportement que pour les attributs,

4. **`type retour`** peut être : un type élémentaire ou un tableau, une classe, rien dans le cas d'un constructeur (page suivante), ou d'une méthode renvoyant le type **`void.`**
5. **`liste paramètres`** peut être : liste de paramètres typés et nommés séparés par des virgules, cette liste pouvant être vide.

Concepts objets de Java

Constructeurs de classe

- **Définition**
  - Un constructeur est une méthode particulière appelée dans la phase d'instanciation d'un objet par Java :

```
Cercle c = new Cercle(20, 20, 10.0);

Cercle c2 = new Cercle();
```

  - Un constructeur porte le nom de la classe,
  - il ne possède pas de type de retour,
  - il peut possèder un nombre quelconque de paramètres,
  - une classe peut avoir plusieurs constructeurs dont le nombre et le type de paramètres diffèrent (surcharge).

```
public Cercle(int cx, int cy, float r){
   x = cx;
   y = cy;
   rayon = r;
}
```

Concepts objets de Java

Constructeurs de classe

- **Comportement et rôle du constructeur**
  - Le constructeur est appelé au moment de la création de l'objet (instanciation). Sa finalité est d'initialiser cet objet en fonction de paramètres fournis par l'utilisateur.
  - Si vous ne créez pas de constructeur, Java en créera un pour vous de manière implicite, n'acceptant aucun paramètre.
  - Attention, si vous créez au moins un constructeur acceptant des paramètres et aucun constructeur n'acceptant pas de paramètres, votre classe n'a plus de constructeur par défaut !
  - Si *en première instruction* de votre constructeur, vous n'appelez pas un constructeur de la classe père (avec **`super(...)`**, voir plus loin), le constructeur par défaut de la classe père est appelé.
  - Attention, si la classe père n'a pas de constructeur par défaut, cela produit une erreur de compilation.
  - La libération des ressources mémoire est prise en charge par le garbage-collector. En revanche, il existe une méthode optionnelle finalize() qui est appelée durant la phase de destruction de l'objet, si elle existe. Cette méthode est optionnelle et n'est utile que dans le cas de la gestion de ressources spécifiques telles fichiers, sockets, etc...

Concepts objets de Java

Instanciation

- **Pour instancier un objet**

1. Déclarer une variable de la classe que l'on désire instancier. Cette variable est une référence sur l'objet que l'on va créer. Java y stockera l'adresse de l'objet.

```
Cercle c;
```

2. Allouer la mémoire nécessaire à l'objet et l'initialiser. cette opération se fait par l'intermédiaire de l'opérateur new auquel on passe l'appel au constructeur de la classe désirée.

```
c = new Cercle(23, 24, 40.0);
```

Remarque : on peut faire les deux étapes en même temps :

```
Cercle c = new Cercle(23, 24, 40.0);
```

- **L'objet est détruit lorsque plus personne ne le référence (sortie d'un bloc de visibilité, etc...)**

Concepts objets de Java

Utiliser un objet

- **Envoi de message**

```
Cercle c = new Cercle(0,0,10.0);
c.modifierRayon(50.0);
c.dessiner();
```

- Envoi de message : invocation d'une méthode sur un objet particulier. Syntaxe :

  `[receveur.] méthode(liste paramètres);`

- **`receveur`** = objet sur lequel on invoque la méthode. L'objet doit exister (avoir été alloué par **`new`** préalablement).
- **`receveur`** est optionnel si l'envoi de message est écrit dans le corps d'une méthode de la classe du receveur.
- **`méthode`** = nom de la méthode invoquée.

  Cette méthode doit faire partie de l'interface publique du receveur, si l'invocation est extérieure à la classe.
- **`liste paramètres`** est une liste de paramètres effectifs correspondants aux paramètres formels spécifiés dans la déclaration de la méthode.

Concepts objets de Java

Utiliser un objet

- **Accès à un attribut**

```
// Si x et y étaient des attributs
// publics de Cercle

Cercle c = new Cercle();

c.x = 100;
c.y = 100;
```

- Syntaxe d'accès à un attribut :

```
[receveur.] nom_attribut;
```

- receveur est optionnel si l'accès à l'attribut est réalisé dans le corps d'une méthode de la classe du receveur.
- Peut être utilisé en partie gauche ou droite d'une affectation

```
c.x = 100;
val_x = c.x;
```

- L'attribut doit être public si on y accède de l'extérieur de la classe.

Il est préférable de fournir des méthodes d'accès aux attributs afin de respecter les principes d'encapsulation et de ne pas rendre le code utilisateur trop dépendant des types des attributs ni de risquer de voir un code utilisateur compromettre l'intégrité de l'objet

Concepts objets de Java

Composition d'objets

- **Agrégation**

```
public class Cercle {

  private Point origine;
  private float rayon;

  public Cercle(int cx, int cy,
                        float r) {
     origine = new Point(cx, cy);
     rayon = r;
  }
}
```

Avec :

```
public class Point {

  private int x;
  private int y;

  public Point(int px, int py){
     x = px;
     y = py;
  }
}
```

Concepts objets de Java
liens entre objets

- **Agrégation**
  - Une instance de **`Cercle`** est propriétaire de son instance de **`Point`** référencée par l'attribut **`origine`** (mais elle peut éventuellement la partager).

    Le constructeur de **`Cercle`** doit prendre en charge la construction et l'allocation de l'attribut **`origine`**, en appelant le constructeur de sa classe.
  - **`origine`**, de la classe **`Point`**, est un sous-objet du **`Cercle`**.
  - Lorsque le garbage-collector détruira le **`Cercle`**, il détruira également tous les sous-objets propriété exclusive du **`Cercle`**, donc le **`Point origine`**. La raison en est uniquement le fait que la destruction de l'objet **`Cercle`** entraînera la perte de l'unique référence sur l'objet **`Point origine`**.

Concepts objets de Java

liens entre objets

- **Associations et usage de this**
  - Supposons qu'un **`Cercle`** possède une référence sur la Fenêtre graphique à laquelle il est attaché

```
public class Cercle {

    private Point origine;
    private float rayon;
    private Fenetre fenetre = null;
    ...
    public void placer(Fenetre f) {
        if(fenetre != null)
            fenetre.enlever(this);
        fenetre = f;
        f.ajouter(this);
    }
}
class Fenetre {
    private Liste cercles;
    ...
    public void ajouter(Cercle c){...}
    public void enlever(Cercle c){...}
}
```

```
Fenetre principale = new Fenetre(...);
Cercle c1 = new Cercle(0, 0, 10.0);
Cercle c2 = new Cercle(100, 100, 10.0);

c1.placer(principale);
c2.placer(principale);
```

`Devient this dans le corps de placer()`

Concepts objets de Java

liens entre objets

- **Association et usage de this**
  - Le **`this`** permet à un objet d'obtenir une référence sur lui-même. Ceci est très utile lorsqu'un objet a besoin de laisser “une carte de visite” à un autre objet pour lui dire “c'est moi qui t'ai contacté, voici par quelle référence tu peux m'utiliser”.

- **Avec l'exemple, voici ce qui se produit :**

1. **`Cercle`** définit un attribut **`fenetre`** qui va permettre à chaque instance de référencer sa fenêtre (celle où il est affiché).
2. Dans ce cas, un **`Cercle`** n'est pas propriétaire de la fenêtre dans laquelle il se trouve. Le constructeur de **`Cercle`** n'est pas responsable de la création de la fenêtre. Celle-ci est créée indépendamment en dehors de la classe **`Cercle`**.
3. L'attribut **`fenetre`** de **`Cercle`** est mis à jour par la méthode **`placer()`**
4. **`Fenetre`** possède une liste de Cercles. La méthode **`placer()`** gère la suppression du cercle dans l'ancienne fenêtre et l'ajout dans la nouvelle, en invoquant **`ajouter()`** et **`enlever()`** de la classe **`Fenetre`**. Ces méthodes prennent en paramètre un **`Cercle`**. Celui-ci n'est autre que le receveur de la méthode **`placer()`**, référencé par **`this`**.

Concepts objets de Java

Exemple de this explicite

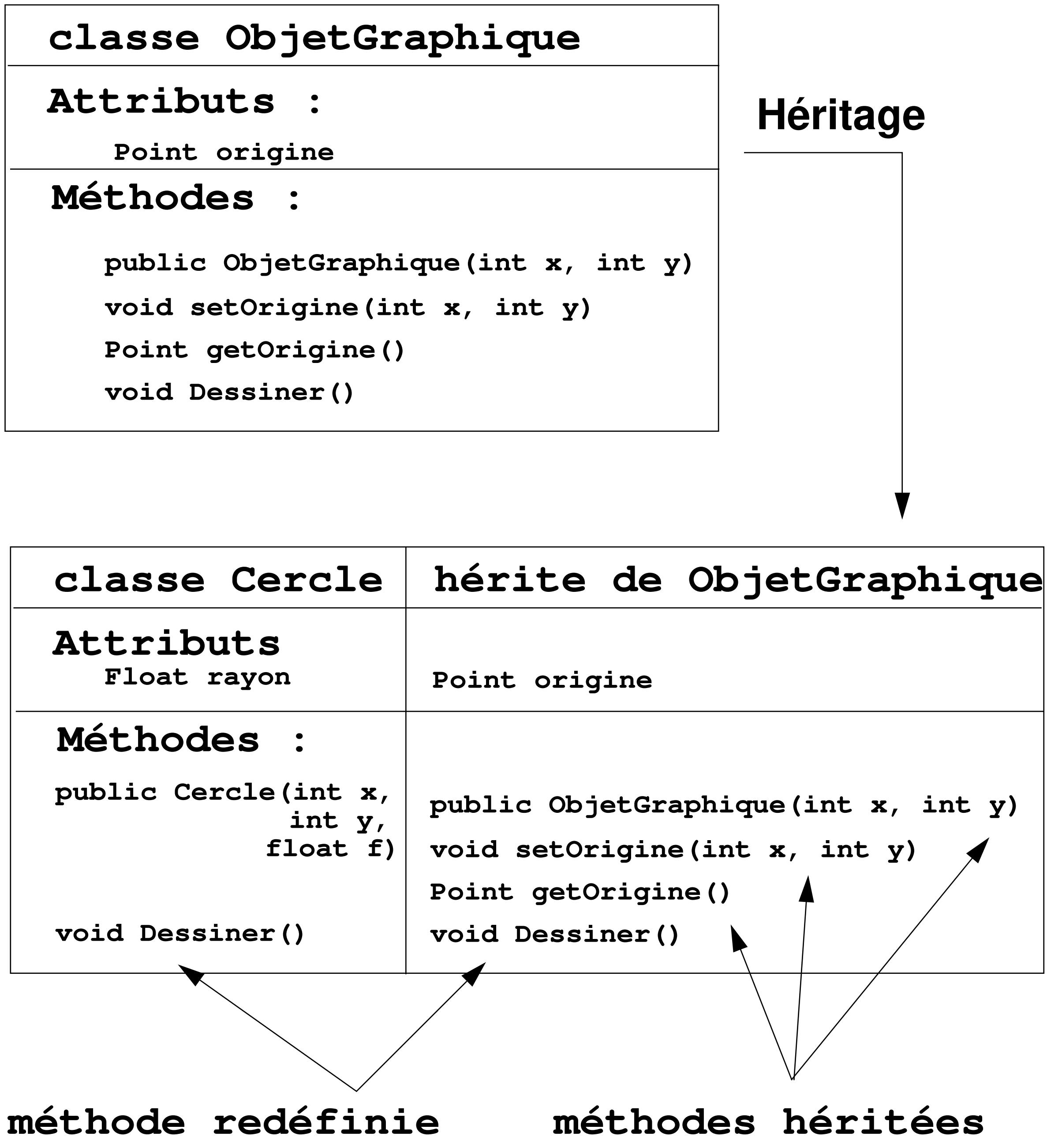
- **Une utilisation très courante de this :**

```
public class Cercle {
    int rayon;

    public Cercle(int rayon) {
        this.rayon = rayon;
    }
}
```

Héritage en Java

Introduction

- **Dans le prototype de la classe à l'aide du mot-clé `extends`**
- **Une classe ne peut hériter que d'une seule classe**



Héritage en Java

Introduction

- **Syntaxe**

```
class <dérivée> extends <base>
```

- **Permet de définir un lien d'héritage entre deux classes**

`<base>` est le nom de la super-classe ou classe de base, **`<dérivée>`** est le nom de la sous-classe ou classe dérivée.

Le lien d'héritage est obligatoirement simple. L'héritage multiple est cependant partiellement réalisable à l'aide du concept d'interface qui sera étudié plus tard.

```
class ObjetGraphique {
    private Point origine;
    public ObjetGraphique(int x, int y) {
        origine = new Point(x, y);
    }
    ...
}
```

```
class Cercle extends ObjetGraphique {
    private float rayon;
    public Cercle(int x, int y, float r) {
        super(x, y);
        rayon = r;
    }
    ...
}
```

Héritage en Java
Introduction

- **Attributs privés**

  L'attribut **`origine`** de **`ObjetGraphique`** n'est pas accessible dans le code de **`Cercle`** car déclaré en privé. Cacher ainsi les attributs des classes est recommandé car le code demeure plus portable.

- **Construction des classes**

  - Lorsqu'on construit une instance de **`Cercle`** on obtient un objet dont une partie est construite grâce à la définition de **`Cercle`** et une partie grâce à la définition de la super-classe **`ObjetGraphique`**.
  - Il faut donc que le constructeur de la sous-classe fasse appel au constructeur de la super-classe. Cela se fait en appelant la méthode **`super()`**.
  - Si on n'appelle pas le constructeur de la super-classe, le constructeur par défaut est utilisé.

- **Lorsque dans un constructeur on invoque un constructeur de la classe mère -avec super() -, *ce doit nécessairement être dans la première instruction !***

Héritage en Java

Redéfinition et surcharge

- **Redéfinition et super**

```
class ObjetGraphique {
   Point origine;
   ...
   public void afficher(){
      origine.dessiner();
   }
   ...
}

class Cercle extends ObjetGraphique {
   ...
   public void afficher() {
      super.afficher();
      ...
   }
   ...
}
```

- **Surcharge**

```
class Cercle extends ObjetGraphique {
   private float rayon;
   ...
   public Cercle(int x, int y, float r)
{...}

   public Cercle() {...}
   ...
}
```

Héritage en Java

Surcharge et constructeurs

- **Un constructeur peut utiliser le code d'un autre par appel explicite à `this(...)` en première instruction**

```
class Cercle extends ObjetGraphique {
  private float rayon;
  ...
  public Cercle(int x, int y,
                          float r{
     ...
  }

  public Cercle() {
     this(50, 50, 10.0f);
     ...
  }
  ...
}
```

Héritage en Java
Redéfinition et surcharge

- **Redéfinition et super**
  - la redéfinition d'une méthode d'une classe consiste à fournir dans une sous-classe une nouvelle implémentation de la méthode. Cette nouvelle implémentation masque alors complètement celle de la super-classe
  - La re-déclaration doit être strictement identique à celle de la superclasse : même nom, même paramètres, type de retour co-variant.
  - Grâce au mot-clé **`super`**, la méthode redéfinie dans la sous-classe peut réutiliser du code écrit dans la méthode de la super-classe, qui n'est plus visible autrement.
  - **`super`** à un sens uniquement dans une méthode (comme le mot-clé **`this`**).

- **Surcharge**
  - Le mécanisme de surcharge permet de réutiliser le nom d'une méthode déjà défini, pour une autre méthode qui en différera par ses paramètres.
  - La méthode surchargée doit conserver la même "intention sémantique" (souhaitable mais pas forcé).
  - La surcharge peut se faire sur une méthode définie localement ou héritée.

Héritage en Java

Visibilité et héritage

- En plus de public et private, de nouveaux mots-clés permettent de régler la visibilité au sens de l'héritage

- protected

```
class ObjetGraphique {
   protected Point origine;
   ...
}

class Cercle extends ObjetGraphique {
   protected float rayon;
   ...
}
```

- friendly (par défaut)

- final

```
class ObjetGraphique {
    Point origine; // friendly
   ...
}

class Cercle extends ObjetGraphique {
   final float rayon = 10.0;
   ...
}
```

Héritage en Java

Visibilité et héritage

- **Friendly (par défaut)**
  - **`public`** vis-à vis du package
  - **`private`** vis-à-vis du reste.
  - Un source utilisateur ne faisant pas partie du package ne pourra pas référencer des attributs ou méthodes **`friendly`**. Bon pour l'encapsulation.

- **Protected**
  - **`public`** vis-à-vis de la classe, de ses sous-classes ET du package, **`private`** vis-à-vis du reste,
  - *dans des classes qui ne sont pas dérivées*, les attributs et les méthodes protégées ne sont pas manipulables,
  - *dans une classe dérivée* une méthode protected héritée est redéfinissable,
  - *dans une classe dérivée*, une variable **`protected`** héritée n'est manipulable que
    - directement ou avec super.attribut,
    - au travers d'objets instanciés à partir de classes dérivées de la classe mère, pas directement au travers d'une instance de la classe mère. Voir exemple page suivante
  - on peut dire que **`public`** donne l'interface pour les "utilisateurs" et **`protected`** décrit le matériel que peut utiliser en plus le "développeur de sous-classes".

Héritage en Java

Visibilité et héritage

- **Un exemple :**

```
Package P1;

public class A {

  protected int x;

  protected f() {};
}
```

```
Package P2;

public class B
        extends A {

  public test() {

    a = new A();
    b = new B();
    c = new C();
    d = new D();

    this.x = 0;
    b.x = 0;
    c.x = 0;

    a.x = 0;

    this.f();
    b.f();
    c.f();

    a.f();

    d.x = 0;

  }
```

```
}
```

```
Package P3;

public class C
    extends B {
}
```

```
package P4;

public class D
      extends A {
}
```

Marche pas !

Marche !

- **`a.f()` et `a.x` ne sont pas acceptés car `a` n'est pas une instance d'une classe dérivée de la classe courante, c'est à dire une instance de la classe `B` (polymorphisme: les instances de `C` sont aussi des instances de `B`)**

- **Mais `super.x` marche (car `x` et `super.x`, c'est la même chose ici) ! Bien que `A` soit égal à `super` !**

Héritage en Java
Visibilité et héritage

- **final**
  - Un attribut **`final`** doit être initialisé à la déclaration et est invariable. Il ne peut être surchargé, il devient une constante.
  - Un paramètre d'une méthode peut être **`final`**. Dans ce cas, *il est assimilé à une constante* dans le corps de la méthode. Aucun affectation ne pourra être effectuée sur l'objet, il est inmodifiable.
  - Une méthode **`final`** ne peut être surchargée. Gros gain de performance à l'exécution !

    L'adresse de la méthode est connue à la compilation.

Héritage en Java

Classe abstraite

- **Utilité**
  - Définir des concepts incomplets, qui devront être implémentés dans les sous-classes.
  - Factoriser le code.
- **Exemple**

```
abstract class ObjetGraphique {
   private Point origine;

   public ObjetGraphique(int x, int y){
      origine = new Point(x, y);
   }

   abstract public void dessiner();
}

class Cercle extends ObjetGraphique {
   private float rayon;

   public Cercle(int x, int y, float r){
      super(x, y);
      rayon = r;
   }

   public void dessiner() {
      // code d'implémentation...
   }
}
```

Héritage en Java

Classe abstraite

- **Fonctionnement des classes abstraites**
  - Regrouper certaines caractéristiques communes à ses sous-classes et définir un comportement minimal commun.
  - Une classe comportant au moins une méthode abstraite est abstraite. Dans ce cas, le mot-clé abstract doit être ajouté devant la définition de la classe.
  - Une classe abstraite ne peut être instanciée, puisqu'elle n'est pas complètement définie.
  - Le compilateur vérifie que toutes les méthodes abstraites sont implémentées par toutes les sous-classes (sauf si la sous-classe est elle-même abstraite).
  - Il est recommandé de mettre un maximum de code dans une classe abstraite ! C'est ça la factorisation du code !

Héritage en Java
Interface

- **Définition**
  - une classe **`Fenetre`** gère une liste d'instances qui doivent au minimum posséder la méthode **`dessiner()`** pour pouvoir s'afficher dans la fenêtre.
  - On définit ce "comportement minimal" dans une **`interface`**, si on ne veut pas que ce comportement soit lié uniquement à une classe et à ses sous-classes, comme dans l'exemple précédent, avec la classe abstraite **`ObjetGraphique`**.

```
interface Dessinable {
   public void dessiner();
}
```

- **Utilisation**
  - Pour pouvoir mettre un **`Cercle`** dans une **`Fenetre`** et que celle-ci soit en mesure de le dessiner, il faut :

```
class Cercle extends ObjetGraphique
                implements Dessinable {

   public void dessiner() {
      // code du dessin de l'objet
   }
}
```

  - Un cercle peut maintenant être utilisé partout où on attend un objet **`Dessinable`**, puisqu'on est sûr qu'il saura se dessiner.

Héritage en Java
Interface

- **Spécification formelle de classe**
  - Une interface définit un ensemble de services que les classes se réclamant de cette interface devront implémenter (sauf si elles sont abstraites).
  - L'implémentation d'une interface est libre.
  - Une classe peut implémenter plusieurs interfaces.
  - Les interfaces peuvent se construire par héritage.
  - Une interface peut être utilisée comme un type.
  - Une interface permet à une classe de s'intéresser uniquement à certaines caractéristiques d'un objet.
  - Lorsqu'on examine une classe implémentant une ou plusieurs interfaces, on est sûr que le code d'implémentation est dans le corps de la classe. Excellente localisation du code (défaut de l'héritage multiple, sauf si on hérite que de classes abstraites)

- **Une interface *peut être vue* comme une classe abstraite dont toutes les méthodes sont abstraites, mais le mot clé abstract n'apparaît jamais (exemple page suivante).**

Héritage en Java

Interface

- **En pratique, dans le code, une interface ne peut contenir que des constantes (`static final`) et des méthodes publiques non implémentées**

```
interface Nommable {
    public String getName();
    public void setName(String s);
}
```

- **Attention, ne pas confondre le concept d'interface dont nous parlons ici, avec l'interface publique d'une classe, qui de façon globale, indique l'ensemble des attributs et méthodes publiques d'une classe.**

Héritage en Java

Interface et polymorphisme

- **Exemple**

```
public class Cercle extends ObjetGraphique {
    ...
    public void dessiner() {
        // code du dessin de l'objet
    }
}
public class Rect extends ObjetGraphique {
    ...
    public void dessiner() {
        // code du dessin de l'objet
    }
}
```

```
public class Fenetre {
   private Vector figures;

   public void ajouter(Dessinable o) {...}
   public void enlever(Dessinable o) {...}

   public void dessiner() {
      int n = figures.size();
      Dessinable o;

      for(int i = 0; i < n; i++) {
         o = (Dessinable)
                    (figures.elementAt(i));
         o.dessiner();
      }
   }
}
```

- **Très utile lorsqu'on manipule des Vector (listes d'Objects)**

Héritage en Java

Interface et polymorphisme

- **Polymorphisme et interfaces**

```
public class ObjetGraphique
                implements Dessinable {

   private Point origine;
   private Fenetre fenetre;
   ...
   public void placer(Fenetre f) {
      if(fenetre != null)
         fenetre.enlever(this);

      fenetre = f;
      f.ajouter(this);
   }
}
```

  - Les classes **`Cercle`** et **`Rect`** sont des **`ObjetGraphique`** qui implémentent l'interface **`Dessinable`**.
  - La **`Fenetre`** est capable de représenter graphiquement tout objet conforme à l'interface **`Dessinable`**.
  - Pour cela, elle gère une liste d'objets de type **`Dessinable`**, grâce à un **`Vector`** et aux méthodes **`ajouter()`** et **`enlever()`**. **`Vector`** est une classe du package **`java.util`** permettant de gérer des listes d'objets quelconques.

Héritage en Java

Attributs et méthodes statiques (dits de classe)

- **La staticité permet de définir**
  - Des attributs partagés par toutes les instances de la classe.
  - Des méthodes non liées à une instanciation.

```
public class ObjetGraphique
             implements Dessinable {

  private Point origine;
  static public int nbObjets = 0;
  ...
  public ObjetGraphique(int x, int y) {
     origine = new Point(x, y);
     nbObjets++;
  }

  static public int nombre() {
     return nbObjets;
  }
}
```

  - **`nbObjets`** est un attribut statique, visible par tous les objets instanciés.

```
Cercle c = new cercle(0, 0, 10.0);
ObjetGraphique f = new
             ObjetGraphique(10, 20);
System.out.println(Figure.nombre());
System.out.println(c.nbObjets);
```

Héritage en Java

Attributs et méthodes statiques

- **Principe de la staticité**
  - La staticité permet de définir des attributs ou des méthodes qui ne sont pas liés à une instance d'objet. On les référence par rapport à la classe. on dispose alors...
  - Pour les attributs : d'attributs uniques, partagés par tous les objets en fonction du rang de visibilité. Par exemple un attribut statique privé ne sera partagé que par les instances de la même classe.

```
System.out.println(c.nbObjets);
```

  - Pour les méthodes : de méthodes dont l'exécution se lance à partir de la classe.

```
System.out.println(Figure.nombre());
```

  - Ici **`out`** est une variable **`static`** de la classe **`System`**, elle même dérivée de **`PrintStream`**. **`println`**() est une méthode public de la classe **`PrintStream`**.
  - **`System`** et **`PrintStream`** seront étudiées en détail dans le chapitre consacré aux entrées/sorties.

Héritage en Java
Attributs et méthodes statiques

- **Attributs de classe**

```
static [visibilité] type nom [ = expression] ;
```

- Attaché à une classe donnée. Est à la classe ce qu'un attribut est à un objet.
- Utilisable et accessible en fonction du rang de visibilité qui lui est affecté.

- **Méthode de classe**

```
static [visibilité] type_retour nom (liste paramètres) ;
```

- Les méthodes statiques ne peuvent accéder qu'à des attributs statiques, en fonction de leur rang de visibilité.
- Ne peuvent pas accéder à une référence this, puisque non liées à une instance d'objet.

Packages
Regroupement de classes

- **Java propose de nombreux packages avec le JDK**
- **java.lang : classes de base du langage**
  - Object,
  - Class, ClassLoader, Runtime, Processus, System.
- **java.util : classes utilitaires**
  - structures de données : Vector, Enumeration, Hashtable, Dictionary, properties, Stack, BitSet.
  - mécanismes de dépendance : Observable, Observer.
  - gestion des dates : Date, Calendar, Gregoriancalendar, Timezone.
  - divers : Random, StringTokenizer, Locale, ResourceBundle, etc...
  - Evénements : Eventlistener, EventObject,
- **java.io : entrées/sorties**
  - Nombreuses classes... voir chapitre dédié.
- **java.math :**
  - BigDecimal, BigInteger (calculs de grande précision)

Packages

Regroupement de classes

- **java.net : le réseau**
  - URL, Socket, SocketServer, etc...
- **java.awt : librairie graphique**
  - Component, Container
  - Evénements
  - Items graphiques : Button, Menus, etc...
  - Trois autres packages complémentaires : java.awt.image, java.awt.event, java.awt.datatransfer
- **java.applet : classe Applet**
- **java.rmi**
  - Remote Method Invocation, applications distribuées.
- **java.security**
  - Signatures digitales, contrôle d'accès, etc...

Packages
Regroupement de classes

- **Intérêt des packages**
  - Facilite le développement de bibliothèques et de modules autonomes.
  - Les classes d'un même package travaillent conjointement sur un même domaine.
  - Le package offre un niveau de protection supplémentaire pour les attributs et les méthodes grâce aux modes protected et friendly.

- **Organisation**
  - Les classes d'un package se trouvent dans un répertoire décrit par le nom du package.
  - Ainsi les classes du package Java.lang se trouvent dans un sous-répertoire java/lang accessible par la variable d'environnement CLASSPATH.

Packages
Utilisation

- **Exemple : on place en début de fichier utilisateur**

```
import java.util.Vector; // (1)

import java.awt.*; // (2)

import ObjetGraphique; // (3)

class Cercle {
   // (4)
   private java.awt.Point origine;
   ...
}
```

1. Utilisation de la classe Vector du package java.util.
2. Importation de toutes les classes du package java.awt.
3. Importation d'une classe définie localement et n'appartenant à aucun package.
4. Accès à une classe en utilisant le nom complet du package.

Packages

Création

1. Ajouter dans chaque fichier “.java” composant le package la ligne :

```
package <nom_du_package>;
```

2. Compiler les sources du package avec l’option -d (pour destination) si on ne désire pas que les fichiers “.class” se trouvent au même endroit que les fichiers source.

```
javac -d $HOME/nom_du_package <classe>.java
```

- Lorsque le source ne contient pas de nom de package, la classe est placée dans un package par défaut, sans nom. Son répertoire est celui du fichier source.

- Pour accéder au package nouvellement crée, ajouter son chemin d’accès dans la variable CLASSPATH.

Les classes intérieures (*inner classes*)

Une classe définie dans une autre classe ?

- **Nouveau dans le JDK 1.1. Une classe intérieure est une classe définie à l'intérieur d'une autre classe.**

- **Pourquoi faire ?**
  - Un objet d'une classe intérieure peut accéder à l'implémentation de l'objet qui l'a créé, y compris à ses attributs **`private.`**
  - Il existe des classes intérieures anonymes (**`anonymous`**) qui sont très pratiques lorsqu'on veut définir des callbacks "au vol".
  - Les classes intérieures sont cachées des autres classes d'un même package (sauf si elles sont **`static`**, nous y reviendrons).
  - Les classes intérieures sont très pratiques lorsqu'on écrit des programmes qui utilisent le nouveau modèle de gestion des évènements du JDK 1.1

Les classes intérieures

- **Nouveauté dans le JDK 1.1**
  - En Java1.0 toutes les classes sont top-level...
  - ...et il n’y a que des classes au top-level !

- **Les classes intérieures du JDK 1.1 seront souvent liées aux...**
  - Evénements,
  - JavaBeans.

- **...mais elles peuvent être employées n’importe où.**

- **Améliorent la cohérence de la syntaxe**
  - Régularisation des règles du contexte (scope),
  - généralisation aux classes.

```
public class Toto {
   ...
   class Titi {
      ...
   }
   ...

}
```

Les classes intérieures
Plusieurs types

- **Une classe intérieure peut être de plusieurs types**
- **Top-level :**
  - Elle peut faire partie d'un package (comme en java 1.0).
  - Elle peut être déclarée en static et donc devenir un membre statique d'une autre classe, comparable aux méthodes et variables statiques.
- **Membre d'une autre classe**
  - Dans ce cas, on peut la comparer aux méthodes et variables d'instances.
- **Locale à une autre classe**
  - Comparables aux méthodes et variables locales
- **Anonyme**
  - Comportement particulièrement adaté

Les exceptions

Les Exceptions

- **On ne peut éviter toute erreur**
  - Plus de mémoire libre
  - mauvais URL
  - division par zéro
  - ...
- **Certaines erreurs sont prévisibles**
  - Charger un fichier inexistant
  - Lire une image dans un format inconnu
  - ...
- **Objectifs de ce chapitre**
  - Lister les différents mécanismes de traitement d’erreur
  - Comprendre les principes de programmation par exceptions
  - Savoir lever et traiter une exception
  - Connaître les types d’exceptions que le JDK peut lever
  - Créer de nouvelles classes d’exceptions

Exceptions
L'approche maladroite

- **Code non spécialisé**
  - Traitement d'erreurs mélangé dans le code
- **Renvoi de conditions d'erreur**
  - Comment renvoyer condition et résultat ?
- **Difficile à comprendre**
  - Modélisation des différents types d'erreur ?
- **Difficile à maintenir**
  - Séparer le code de détection d'erreur du code de traitement de l'erreur
  - Comment traiter les erreurs fatales ?

```
code = ouvreFichier(...);
if (code == ERREUR)
   traiteErreur(code);
else {
   ....
```

```
Comment traiter une
erreur fatale ?
Comment reprendre la
suite du code ?Avec
setjump() et longjump()
allez....
```

Exceptions

En java une exception est un objet envoyé à la JVM

- **A la réception :**
  - Arrêt du code
  - Recherche du traitement de l'erreur identifié par un objet de la classe Exception, paramètre du bloc de traitement
  - La recherche s'effectue en local, puis remonte la liste des appelants
  - Aquittement de l'Exception et reprise du code

- **Structures spécialisées try{}, catch{}, finally{}, throw()**

```
try{
    k = Divise(i, 0)
}
catch(Exception e) {
    // traitement de
    // l'exception
}
.....
```

```
// A l'interieur de Divise...
    ...
    i /= j
    // Code jamais exécuté
    ....
    ....
    ...
```

Traitement de l'erreur local

Sortie immédiate du code, création d'un objet de la classe Exception

Exceptions
Bloc try{...}, catch{...}

- **C'est le bloc de traitement des exceptions**
- **Le try{....} délimite la partie du code que vous désirez surveiller**
- **Un ou plusieurs catch(TypeException exception){...} permettent de récupérer la main**
  - Plusieurs catch(){...} pour spécialiser le traitement
  - L'objet de la classe Exception contient de nombreuses méthodes et attributs renseignant l'erreur

```
catch(NullPointerException e){
   ...
}
catch(OutOfMemoryException e) {
   ...
}
finally{
   // on passe toujours ici !
}
etc... suite du code
```

- **Pas comme dans un switch {...} : on ne peut passer que dans un seul catch{}**

Exceptions

Types d'exceptions

Table 1: différents types d'exceptions

Type	**Définir**	**Lever**	**Traiter**	**Déclarer**
Error	NON	NON	OUI	NON
RuntimeException	NON (*)	OUI	OUI	NON
Utilisateur	OUI	OUI	OUI	OUI

- **Error = erreurs imprévues (InternalErrorException)**

- **RuntimeException = exceptions liées à la machine virtuelle (NullpointerException)**

- **Utilisateur = exceptions définies par le programmeur, propres à son application (AlienKilledException)**

Exceptions
RuntimeExceptions

- **ArithmeticException**
  - Erreurs mathématiques, division par zéro, etc...
- **NullPointerException**
- **ClassCastException**
  - Tentative de conversion d’un type vers un autre non valide
- **NegativeArraySizeException**
  - On créée un tableau de taille négative ? Tssst...
- **ArrayIndexOutOfBoundsException**
  - Accès à un élément d’un tableau qui est en dehors des bornes
- **OutOfMemoryException**

Exceptions

Méthodes de la classe Exception

- **toString()**
  - Existe pour presque tous les objets Java

```
catch(ArithmeticException e) {
   System.out.println(“exception : ” +
                  e.toString() + “ rencontrée ”);
}
```

- **printStackTrace()**
  - Imprime l’ensemble des appels ayant conduit à la levée de l’exception
  - Appelée par défaut lors de l’acquittement d’une exception par java

```
catch(ArithmeticException e) {
   e.printSTackTrace()
}
AFFICHE LES INFORMATIONS SUIVANTES :
java.lang.ArithmeticException: / by zero
      at Vector.normalize(Vector.java: 25)
      at Vector.getMatrix(Vector.java: 140)
......
```

Exceptions

Méthodes de la classe Exception

- **A l'aide du mot-clé *throws* dans le prototype d'une méthode**

```
public void methode1() throws [ClasseException] {
    ...
}
```

- **A partir du moment où une méthode lève une exception, deux possibilité d'utilisations dans la méthode appelante**

1. On utilise un bloc try{}

```
public void methode2() {
    try{
        methode1();
    catch(Exception e) {
        // traitement...
    }
}
```

2. La méthode appelante relaye l'exception

```
public void methode2() throws [ClasseException] {
    methode1();
    ...
}
```

Exceptions

Et si je relaye une exception jusqu'à la JVM ?

- Traitement par défaut : affichage de la pile des appels de méthodes
- Acquittement de l'exception
- Cool.... comparer ce mécanisme à des codes binaires par exemple !

Exceptions

Lesquelles déclarer ?

- **Les exceptions telles que ArithmeticException, NullPointerException, ClassCastException, etc... sont toutes des sous-classes de RuntimeException**
- **On n'a pas besoin de déclarer les exceptions de la classe RuntimeException, car elles sont liées au langage (package java.lang)**
- **Toutes les autres exceptions doivent être déclarées**

Exceptions

Créer une classe d'Exception

- **Exception hérite de Throwable**
  - Throwable : être levées et attrapées + méthodes informatives comme printStackTrace(), toString()
- **Il faut sous-classer Exception**
  - Mécanisme permettant de décrire l'exception

```
public class MonException extends Exception {
    private int valeur;

    public MonException(String info, int valeur) {
        super(info);
        this.valeur = valeur;
    }

    public toString() {
    return super.toString() +
                    Integer.toString(valeur);
    }
}
```

- **Penser à modéliser les exceptions en même temps que le reste de l'application**

Les entrées/sorties

Entrées-Sorties

Terrain connu : la classe Java.lang.System

- **Interface avec le système d'exploitation.**
- **3 fichiers standards :**
  - **`System.in`** : un **`java.io.BufferedInput-Stream`**
    - **`public int read()`** : attend la prochaine entrée et retourne son code.
    - **`public long skip(long)`** : se déplace d'un certain nombre d'octets dans le flot. Retourne le nombre d'octets dépassés.
  - System.out : un java.io.PrintStream
    - **`public void print(tout type Java)`**
    - **`public void println(tout type Java)`**
  - System.err : idem
    - idem **`System.out`**

```
import java.io.*;

class MonApplication {

   public static void main(String[] argv){

      int b, nb = 0;

      try {
         // lecture de caractères au clavier
         while((b = System.in.read()) != -1){
            nb++;
            System.out.print(c);
         }
      } catch(IOException e {...}

      System.out.println("nb = " + nb);
```

Entrées-Sorties

Le packageJava.io

- **Ce package fournit des classes permettant de manipuler diverses ressources.**

- **Fichiers (Classe `Java.io.File`)**
  - Classes **`FileInputStream`**, **`FileOutputStream`** et **`RandomAccessFile.`**

- **Mémoire**
  - **`BufferedInputStream`** et **`BufferedOutputStream`**
    - Lectures bufférisées.
  - **`DataInputStream`** et **`DataOutputStream`**
    - Lectures typées (utile pour lire des fichiers dont on connait la nature exacte : 3 doubles et un entier sur chaque ligne par exemple...)
  - **`PushBackInputStream`**
    - Buffer d'un octet dans lequel on peut remettre le caractère qui vient d'être lu. Similaire au tampon de Pascal.

- **Pipe**
  - **`PipedInputStream`** et **`PipedOutputStream`**
    - Permettent à deux threads d'échanger des données.

- **Lecture filtrée**
  - **`StreamTokenizer`**

Entrées-Sorties

La classe Java.io.File

- **Cette classe fournit de nombreuses méthodes pour gérer des fichiers.**
- **Nommage**
  - `String getName(), String getPath(), String getAbsolutePath(), String get-Parent(), boolean renameTo(File newName)...`
- **Droits d'accès**
  - `boolean exists(), boolean canWrite(), boolean canRead()...`
- **Autres informations**

  `long length(), long lastModified(), boolean delete()...`

```
File f = new File("toto.txt");

System.out.println("toto.txt : " + f.getName() +
                   " " + f.getPath());

if(f.exists()){

   System.out.println("fichier existe, droits = "+
      f.canRead() + " " + f.canWrite() + " " +
      f.length());

   ...
```

Entrées-Sorties

La classe Java.io.File

- **Répertoires**
  - `boolean mkdir(), String[] list()`

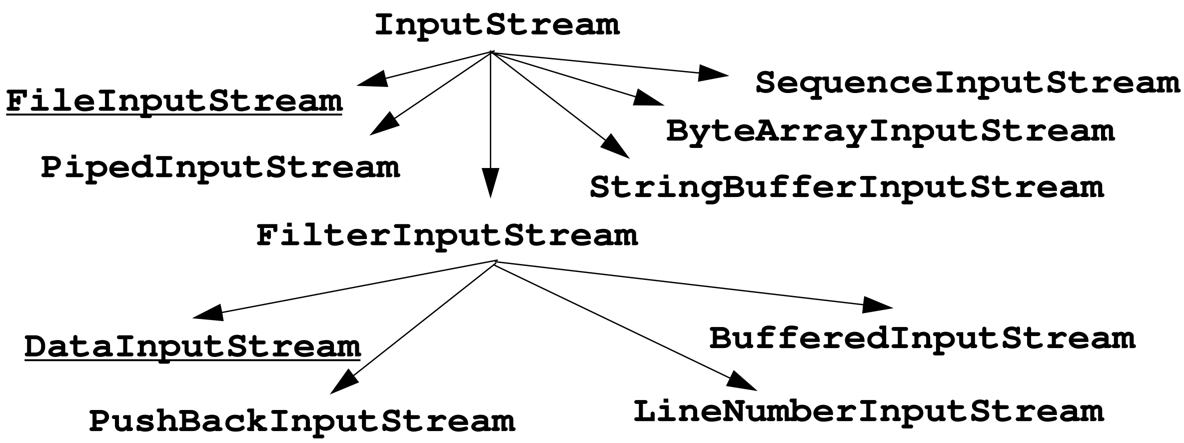
```
File f = new File("/u/I3S/buffa", ".emacs");
// ou bien
File home = new File("/u/I3S/buffa");
File f = new File(rep, ".emacs");
```

Entrées-Sorties

La classe Java.io.InputStream

- **Classe abstraite. Fournit un constructeur par défaut + ensemble de méthodes, dont `read()` que toute sous-classe doit implémenter.**

- **C'est la classe de base de tous les flots d'entrée**



- **Infos sur quelques sous-classes**
  - **`SequenceInputStream`** : permet de travailler avec une énumération d'InputStreams comme s'il s'agissait d'un seul.
  - **`StringBufferInputStream`** : lecture de chaînes de caractères.
  - **`ByteArrayInputStream`** : lecture de tableaux d'octets.
  - **`FilterInputStream`** : encapsule une instance d'InputStream et fournit des méthodes de filtrage.

Entrées-Sorties

La classe Java.io.FileInputStream

- **Lecture de caractères ou d'octets sur un fichier.**
- **Fermeture du fichier implicite (garbage collecting) ou par la méthode `close()`.**
- **Constructeurs**
  - `public FileInputStream(String finename)`
  - `public FileInputStream(File file)`
- **Méthodes (renvoient -1 en fin de fichier)**
  - `int read()` : lit un octet.
  - `int read(byte[] taboctets)` : remplit taboctets si possible. taboctets doit être alloué avant.
  - `int read(byte[] taboctets, int offset, int nb)` : lit nb octets et les met dans taboctets à partir de l'indice offset.

Entrées-Sorties

La classe Java.io.FileInputStream

```
FileInputStream fis;
byte[] b = new byte[1024];

// ouverture du fichier
try {
   fs = new FileInputStream("/u/I3S/buffa/.emacs");
} catch (FileNotFoundException e) {...}

// lecture des données
try {
   int i = fis.read(b);
} catch (IOException e) {...}

// utilisation des caractères lus
String s = new String(b, 0);
```


Entrées-Sorties

La classe Java.io.DataInputStream

- **Lecture de données typées**
  - Portabilité (un entier est de même taille partout).
- **Classe de “spécialisation”**
  - Possibilité de transformer en **`DataInputStream`** n’importe quel **`InputStream.`**
- **Constructeur**
  - **`public DataInputStream(InputStream)`**
- **Méthodes de lecture**
  - **`byte readByte(), short readShort(), char readChar(), int readInt(), float readFloat()...`**
  - Mais aussi **`readLine()`**, avec la fin de la ligne marquée par \n, \r, \r\n ou EOF.
  - ...
- **Fermeture de flot**
  - La fermeture d’un **`dataInputStream`** entraîne la fermeture de l’**`InputStream`** original.

Entrées-Sorties

La classe Java.io.DataInputStream

```
// Ouverture d'un fichier
FileInputStream fis;
fis = new FileInputStream("ball.obj");
// spécialisation !
DataInputStream dis = new DataInputStream(fis);
String ligne = new String();
// lecture des données
while((ligne = dis.readLine()) != null){
   System.out.println(ligne+"\n");
}
// fermeture du fichier. Ferme aussi fis !
dis.close()
```

Entrées-Sorties
InputStreams et URL

- **Il est possible d'obtenir un InputStream à partir d'un URL**
  - Créer un objet de la classe URL,
  - obtenir un stream sur cet URL...
  - ...ou ouvrir une connexion sur l'URL et obtenir un stream à partir de la connexion.

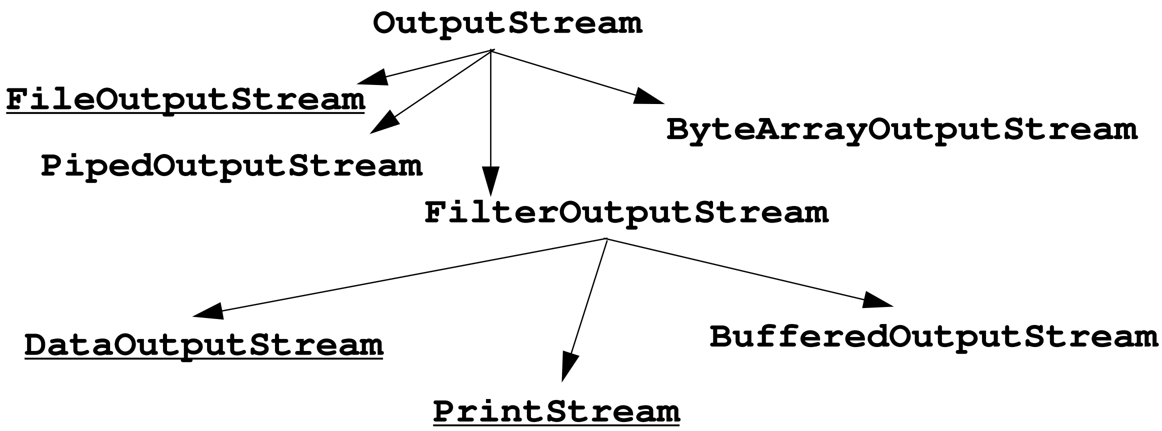
- **Exemples**

```
InputStream is;
DataInputStream dis;
URL mon_url = new URL(getDocumentBase(),
                                        ".emacs");
is = mon_url.openStream();
dis = new DataInputStream(is);
while((String line = dis.readLine()) != EOF){
   System.out.println(line + "\n");
}

// on aurait pu remplacer
// is = mon_url.OpenStream() par
// is = mon_url.openConnection().getInputSTream()
```

Entrées-Sorties

java.io.OutputStream

- **Classe (abstraite) de base de tous les flots de sortie**



- **PrintStream : sorties de données typées autres que des bytes**
  - **`System.out`** est un PrintStream.
- **Méthodes fournies**
  - **`close()`**
  - **`flush()`**
  - **`write()`** : écriture d'octets
    - Doit être implémenté par toute sous-classe de OutputStream.

Entrées-Sorties

java.io.FileOutputStream

- **Ecriture de caractères dans un fichier**
  - Ouverture du fichier à la construction
    - Le sécurity manager est appelée à chaque ouverture d'un fichier.
  - Méthodes d'écriture d'octets.
  - Fermeture explicite avec **`close()`** ou implicite.

- **Constructeurs**
  - **`public FileOutputStream(File)`**
  - **`public FileOutputStream(FileDescriptor)`**
  - **`public FileOutputStream(String)`**

- **Méthodes**
  - **`public void write(int)`**
  - **`public void write(byte [])`**

```
FileOutputStream fos =
                    new FileOutputStream("toto");

String chaine = new String("Coucou c'est moi");
int longueur = chaine.length();
byte[] buffer = new byte[l];
chaine.getBytes(0, longueur - 1, buffer, 0);

for(int i = 0; i < longueur; i++)
   fos.write(buffer[i]);
```

Entrées-Sorties

java.io.BufferedOutputStream

- **Ecriture sur un flot de sortie bufférisé**
  - Spécialisation à partir de n'importe quel flot de sortie.
  - Ecriture sur disque explicite par **`flush()`** ou **`close()`**.

- **Constructeurs**
  - **`BufferedInputStream(OutputStream)`**
  - **`BufferedOutputStream(OutputStream, int size)`**

- **Méthodes**
  - **`write(int)`**
  - **`write(byte [], int offset, int length)`**
    - Construction en précisant la taille du buffer.
  - **`flush()`**
  - **`close()`**

Entrées-Sorties

java.io.BufferedOutputStream

```
// Ouverture d'un fichier pour écriture en mode
// bufférisé

FileOutputStream fos =
             new FileOutputStream("toto.txt");

BufferedOutputStream bos =
                new BufferedOutputStream(fos);


// Préparation des données à écrire

String chaine = "coucou c'est moi";

int longueur = chaine.length();

byte[] buffer = new byte[longueur];

chaine.getBytes(0, longueur - 1, buffer, 0);


// Ecriture

for(int i = 0; i < longueur; i++)

   bos.write(buffer[i]);

bos.write("\n");

bos.close();
```

Entrées-Sorties

java.io.DataOutputStream

- **Ecriture de données typées**
  - Portabilité (un entier est de même taille partout)
- **Classe de “spécialisation”**
  - Possibilité de transformer en **`DataOutput-Stream`** n’importe quel **`OutputStream`**. Conseillé : prendre un **`BufferedOutputStream`**
- **Constructeur**
  - **`public DataOutputStream(OutputStream)`**
- **Méthodes d’écriture**
  - **`byte writeByte(), short writeShort(), char writeChar(), int writeInt(), float writeFloat(), writeDouble(), writeBytes(), writeChars(), writeU-TF()`**
- **Fermeture de flot**
  - Avec **`close()`**
  - La fermeture d’un **`DataOutputStream`** entraîne la fermeture de l’**`InputStream`** original.

Entrées-Sorties

java.io.DataOutputStream

```
FileOutputStream fos = new
                    FileOutputStream(”toto.txt”);
BufferedOutputStream bos =
                    new BufferedOutputStream(fos);
DataOutputStream dos = new DataOutputStream(bos);
dos.writeChars(“Position “);
dos.writeDouble(10.0);
dos.writeDouble(12.0);
dos.writeDouble(7.0);
dos.writeChars(“\n”);
...
```

Entrées-Sorties

java.io.RandomAccessFile

- **N'entrent pas dans le cadre de ce cours...**

Entrées-Sorties

Flots de caractères : nouveau dans le JDK 1.1

- **La version 1.1 du JDK introduit la notion de “flots de caractères” (Character Streams).**
- **Dans les anciennes version, les entrées/sorties ne travaillaient qu’avec des bytes, au travers des classes `InputStream` et `OutputStream`.**
- **Les flots de caractères sont similaires mais au lieu de travailler sur des données de 8 bits, elles manipulent des caractères au format unicode (16 bits).**
- **Ceci est réalisé au travers de deux classes et de leurs sous-classes: les classes `Reader` et `Writer`, qui supportent globalement les mêmes opérations que les classes `InputStream` et `OutputStream`.**
- **Les principales fonctionnalités existants sur les flots d’octets sont disponibles sur les flots de caractères, au suffixe des noms de classes près**
  - FileReader <-> FileInputStream
  - PushbackReader <-> PushbackInputStream

Entrées-Sorties

Flots de caractères : intérêt ?

- **On peut échanger des données textuelles avec des accents, des caractères bizarres (japonais ?), de manière transparente.**

- **Les classes `InputStreamReader` et `OutputStreamWriter` se chargent de l'encodage et du décodage, car en fait, ce sont quand même des bytes qui circulent.**

- **Plus efficaces que les anciens `InputStream` et `OutputStream` qui sont implémentés pour travailler byte par byte.**

- **Les `InputStreamReader` et `OutputStreamWriter` sont bufférisés en standard.**

- **Les plus utilies : `InputStreamReader, BufferedReader, FileReader, PrintWriter et FileWriter.`**

Entrées-Sorties

Flots de caractères : noms des classes

Table 1: Correspondances entre classes travaillant en entrée sur des flots de caractères et celles travaillant sur des flots de bytes

Flot de caractères	*Description*	*Classe correspondante travaillant avec des bytes*
Reader	Classe abstraite pour les flots de caractères en entrée	InputStream
BufferedReader	Bufférise les entrées, ligne par ligne	BufferedInputStream
LineNumberReader	Garde la trace du nombre de lignes lues	LineNumberInputStream
CharArrayReader	Lit un tableau de caractères	ByteArrayInputStream
InputStreamReader	Transforme un flot de bytes en caractères unicode	Pas d'équivalence
FileReader	Transforme des bytes lus depuis un fichier en caractères unicode	FileInputStream
FilterReader	Classe abstraite pour filtrer des caractères en entrée	FilterInputStream
PushbackReader	Lecture de caractères avec un coup d'avance	PushbackInputStream
PipedReader	Lit depuis un PipeWriter	PipedInputStream
StringReader	Lit depuis une chaîne de caractères	StringBufferInputStream

Entrées-Sorties

Flots de caractères : noms des classes

Table 1: Correspondances entre classes travaillant en sortie sur des flots de caractères et celles travaillant sur des flots de bytes

Flot de caractères	*Description*	*Classe correspondante travaillant avec des bytes*
Writer	Classe abstraite pour les flots de caractères en sortie	outputStream
BufferedWriter	Bufférise les sorties, ligne par ligne	BufferedOutputStream
CharArrayWriter	Ecrit un tableau de caractères	ByteArrayOutputStream
OutputStreamWriter	Transforme un flot de caractères en flot de bytes	(Pas d'équivalence)
FileWriter	Transforme un flot de caractères en fichier de bytes	FileOutputStream
PrintWriter	Ecrit des valeurs et des objets dans un Writer	PrintStream
PipedWriter	Ecrit dans un PipeReader	PipedOutputStream
StringWriter	Ecrit dans une chaîne de caractères	(Pas d'équivalence)

Classes utilitaires, package java.util

Le package java.util

Des fonctionalités importantes...

- **Stocker des objets dans une zone de taille variable (tableaux, piles)**
  - Les classes **`Vector`** et **`Stack`**
- **Associer deux objets à l'aide de tables de hash-code, de dictionnaire et de "tables de propriétésé"**
  - les classes **`HashTable`**, **`Dictionary`** et **`Property`**
- **Découper une chaîne de caractères en mots (tokens)**
  - La classe **`StringTokenizer`**
- **Faire en sorte qu'un objet soit averti lorsqu'un autre objet change**
  - Mécanisme **`Observer/observable`**

Le package java.util
La classe Vector

- **Les tableaux java ne répondent pas forcément à tous les besoins**
  - Lorsqu'on ne sais pas combien d'objets on veut stocker dans le tableau
  - on peut créer un tableau très grand, mais ce n'est pas une méthode très "propre"

- **Solution : la classe `Vector`**
  - Un **`Vector`** se comporte comme un tableau : il contient plusieurs objets (de la classe **`Object`**)
  - On accède à ses éléments à l'aide d'un index
  - Ils grossissent automatiquement lorsqu'il n'y a plus assez de place pour contenir de nouveaux objets
  - Ils proposent des méthodes pour ajouter ou enlever un éléments situé à un endroit quelconque

Le package java.util

Création d'unVector

- **A la création on indique la taille initiale et la manière dont le `Vector` doit grandir lorsqu'il ne peut plus héberger de nouveaux objets**
  - Ces deux informations sont facultatives, mais attention à la perte de performance si on laisse java décider tout seul !

- **Constructeurs**

```
public Vector()
public vector(int initialCapacity)
public Vector(int initialCapacity,
                            int capacityIncrement);
```

- **Si on ne précise pas d'incrément de capacité, le `Vector` double sa taille à chaque dépassement**

- **Il est conseillé d'indiquer une taille initiale**

Le package java.util

classe Vector : ajout et modification des objets

- **Deux manières d'ajouter un élément**

  1. à la fin d'un Vector avec la méthode :

```
public final synchronized void addElement(Object
                                          newElement);
```

  2. entre deux éléments avec :

```
public final synchronized void insertElement(Object
        newElement, int index) throws
                           ArrayIndexOutOfBoundsException;
```

  ...le paramètre **`index`** indique la position dans le **`Vector`** où le nouvel élément doit être inséré.

  Si la position est incorrecte, l'exception **`ArrayIndexOutOfBoundsException`** est levée

- **Remplacer un objet situé à une position donnée par un autre**

```
public final synchronized void setElementAt(Object obj,
        int index) throws ArrayIndexOutOfBoundsException;
```

  - Fonctionne comme insertElement() sauf qu'au lieu que les éléments à droite du Vector se décalent pour laisser la place au nouvel élément, ici l'élément est remplacé.

Le package java.util

classe Vector : accès à un élément

- **Impossible d'utiliser l'indexation comme avec les tableaux**
  - Vector v = new Vector();
  - v.addElement(new PacMan());
  - *v[0].display(); // interdit !*

- **Il faut utiliser des méthodes spécialisées**

```
public final synchronized Object elementAt(int index)
                    throws ArrayIndexOutOfBoundsException;

public final synchronized Object firstElement() throws No
                    SuchElementException;

public final synchronized Object lastElement() throws No
                    SuchElementException;
```

- **On peut tester si un vector est vide avec la méthode `isEmpty()`**

```
public final boolean isEmpty();
```

- **Taille d'un vector**

```
public final int size();
```

Le package java.util

classe Vector : recopie dans un tableau

- **Utiliser copyInto()**

```
public final synchronized void copyInto(Object[] tab);
```

- **Un exemple :**

```
Object tab = new
                Object(myVector.size());

myVector.copyInto(tab);
```

- **Effectuer une action sur tous les éléments d'un Vector**
  - La méthode **`elements()`** de la classe **`Vector`** renvoie un objet de type **`Enumeration`**, qui permet d'accéder aux éléments de manière séquentielle
  - Voir pages suivantes...

Le package java.util

L'interface Enumeration

- **Permet d'énumérer une liste d'objets (ils peuvent donc être hétérogènes)**

- **Le plus souvent utilisé en collaboration avec la classe `Vector`, car la méthode `elements()` de cette dernière retroune une `Enumeration`**

- **Deux méthodes principales**
  - **`public abstract boolean hasMoreElements()`** : renvoie true lorsqu'il reste des éléments à énumérer, false sinon
  - **`public abstract Object nextElement() throws NoSuchElementException`** : retourne le prochain objet et avance d'un cran dans l'énumération

- **Exemple d'utilisation**

```
Enumeration e = myVector.elements();

while(e.hasMoreElements()) {
   Object obj_suivant = e.nextElement();
   // Effectuer des actions sur l'objet
}
```

Le package java.util

L'interface Enumeration

- **Attention, les objets récupérés par `nextEle-ment()` sont de type `Object`, il faudra forcément les caster si on veut invoquer des méthodes spécialisées**

```
Vector v = new Vector();

v.addElement(new Personne("Michel",
"Buffa", 32);
v.addElement(new Personne("Peter",
"Sander", 41);


Enumeration e = myVector.elements();

while(e.hasMoreElements()) {
   Object obj_suivant = e.nextEle-
ment();


   // Effectuer des actions sur l'objet
(Personne)obj_suivant.getName();

}
```

Le package java.util

L'interface Enumeration

- **Pour éviter de caster (ce qu'n'est pas très beau)**
  - Si on ne gère que des objets de la même classe MyClass dans le Vector (donc dans son énumération correspondante) : dériver la classe Vector et ré-implémenter la méthode elements() pour qu'elle renvoie une Enumeration d'objets de type MyClass. Il faudra pour cela ré-implémenter une interface Enumeration.
  - Si on gère des objets de classes différentes mais possèdant tous des méthodes précises (ils implémentent donc une ou plusieurs interfaces), on peut dériver Vector de la même manière pour qu'il gère des objets des interfaces implémentées par les objets.
- **Travail utile que si on a des besoins répétés et très spécialisés.**

Le package java.util

classe Vector : recherche d'un élément

- **On peut utiliser une Enumeration du Vector et parcourir tous les éléments**
  - Fastidieux, peu performant
- **Il existe des méthodes spécialisées**
  -
  - Un objet est-il présent ?

```
public final boolean
                    contains(Object obj);
```

  - A quelle position se trouve un objet donné ?

```
public final int indexof(Object obj);
```

  - Un objet est peut-être présent plusieurs fois, je veux la position de sa première occurence, en partant de la position `startIndex` :

```
public final synchronized int indexOf(Object obj,
                                      int startIndex) throws
                         ArrayIndexOutOfBoundsException;
```

  - Idem, mais avec la dernière occurence :

```
public final int
                lastIndexOf(Object obj);

public final synchronized int lastDIindexOf(Object obj,
                                      int startIndex) throws
                         ArrayIndexOutOfBoundsException;
```

Le package java.util

classe Vector : suppression d'éléments

- **Supprimer tous les éléments**

```
public final synchronized void
                        removeAllElements();
```

- **Supprimer un élément spécifique**

```
public final synchronized boolean
              removeElement(Object obj);
```

  - Attention, le premier appel supprime la première occurence de l'objet.

- **Supprimer un élément à une position donnée**

```
public final synchronized boolean
                removeElementAt(Object obj, int index)
                    throws ;ArrayIndexOutOfBoundsException
```

Les applets

Introduction aux Applets

Objectifs de ce chapitre

- **Apprendre ce qu'est une Applet et ce qui la différencie d'une application,**

- **Etudier les points d'entrée d'exécution d'une Applet et son cycle de vie,**

- **Manipuler les classes de bases nécessaires au codage d'une Applet**

- **Définir les codes HTML nécessaires pour charger et exécuter une Applet dans une page HTML, passage de paramètres depuis la page.**

Qu'est-ce qu'une Applet
Rappels

- **Une Applet est une application java exécutée dans un navigateur Web.**

- **Elle ne possède pas de point d'entrée `main()`, car celui-ci se trouve déjà dans le navigateur.**

- **D'autres points d'entrée existent :**
  - public void init();
  - public void start();
  - public void stop();
  - public void destroy();
  - public void paint(Graphic g);
  - ...

- **Il s'agit d'une application indépendante, interprétée par la machine virtuelle du navigateur.**

- **Le programmeur d'Applets dispose d'un ensemble de classes pour les implémenter.**

Composantes de base d’une Applet

Créer une Applet

- **Il suffit d’hériter de la classe `java.applet.Applet`**
- **Et surcharger les méthodes `init()`, `start()`, etc...**

```
import java.applet.*;

public class MonApplet extends Applet {

   // surcharge des méthodes de notre
   // choix

   public void init() {
      System.out.println("Init appelée
         par le navigateur");
   }

   public void start() {
      System.out.println("Start appelée
         par le navigateur");
   }

   public void stop() {
      System.out.println("Stop appelée
         par le navigateur");
   }

}
```

Composantes de base d'une Applet

La classe Applet

- **Par héritage, la classe MyApplet est devenue une Applet**
  - Ma classe est donc appartient donc à toutes les classes de l'arbre d'héritage de la classe Applet

```
java.lang.Object
  |
  +----java.awt.Component
          |
          +----java.awt.Container
                  |
                  +----java.awt.Panel
                          |
                          +----java.applet.Applet
                                  |
                                  +----MyApplet
```

Composantes de base d'une Applet

La classe Applet

- **De quoi hérite MyApplet :**
  - **`Object`** : la classe de base de tout objet java.
  - **`Component`** : cette classe définit un composant graphique au plus haut niveau.
  - **`Container`** : un composant graphique pouvant inclure d'autre composants (Component). Notre applet est un Component, elle pourra donc contenir des objects graphiques.
  - **`Panel`** : un habillage de Container permettant de "placer" les objets inclus de manière intelligente.
  - **`Applet`** : complémente ce qui précède pour fournir les prototypes des points d'entrée appelés par le navigateur web.
  - **`MyApplet`** : ah, ça dépend de ce qu'on va y mettre !

Les méthodes de la classe Applet
Implémentées dans Applet

- **public void init()**
  - Appelée à l'initialisation de l'Applet par le navigateur. Elle n'est eécutée qu'une seule fois. On met dans init() toutes les opérations qui doivent être réalisées préalablement à l'exécution de l'Applet. par exemple, pour une applet de jeu, il faudra charger tous les graphismes et les sons avant de démarrer l'animation.
- **public void start()**
  - Appelée par le navigateur chaque fois qu'il doit démarrer l'Applet. Soit après l'appel à init(), soit après un appel à stop() et l'arrivée d'un événement approprié.
- **public void stop()**
  - Appelé chaque fois que le navigateur arrete l'exécution de l'Applet (l'utilisateur a changé de page web, il a icônifié le navigateur, etc...) penser à arrêter les tâches de fond, les musiques dans ce cas-là !
  - Lorsque la page web apparaît à nouveau ou que le navigateur est désicônifié, start() est appelée.
- **public void destroy() : avant la mort du navigateur**

Les méthodes de la classe Applet

Méthodes graphiques et classe Graphics

- **Le contexte graphique est indispensable à l'exécution de toute manipulation graphique.**
- **Il est transmis sous la forme d'un objet, instance de la classe Graphics.**
- **Ce contexte est transmis par le navigateur, au travers de méthodes dédiées.**
- **Il n'est valable que le temps d'exécution de la méthode car il n'est qu'une copie du contecte graphique que vous partagez avec le navigateur.**

```
import java.awt.Graphics;
import java.applet.Applet;

public class HelloWorldApplet
                          extends Applet {

     public void paint(Graphics gc) {
         gc.drawString("Hello world!",
                                 50, 25);

     }

}
```

Les méthodes de la classe Applet
Les méthodes graphiques

- **public void repaint()**
  - Provoque le rafraîchissement de votre programme graphique.
  - Provoque l'appel asynchrone de la méthode **`paint()`** par l'AWT.
  - Ne pas la surcharger !!!

- **update(Graphics gc)**
  - Appelée par le navigateur chaque fois qu'un bout de votre application doit être re-dessiné. Cette méthode effectue les opérations suivantes :
    - Dessine un rectangle de la couleur du fond sur la zone à re-dessiner,
    - Appelle la méthode paint()
  - Si vous surchargez cette méthode, n'oubliez pas que par défaut, elle n'est pas vide ! (On peut avoir besoin de la surcharger, pour des besoins de performances graphique)

- **public void paint(Graphics gc)**
  - C'est là que doit se faire les opérations de dessin de votre application. Cette méthode ne doit jamais être appelée par le programmeur, elle est appelée automatiquement par l'AWT (lors d'un repaint(), d'une dé-icônification, etc...)

Les méthodes de la classe Applet
Exécuter une Applet

- Une Applet s'exécute au travers d'un navigateur
- On indique le nom de la classe principale, la taille et les paramètres de l'Applet dans une page HTML

```
...
<applet code="HelloWorld.class"
               width=300 height=200>
<H1>Votre navigateur ne supporte pas
java, désolé !</H1>
</applet>
```

- Quand on développe des Applet, les tester à l'aide de l'AppletViewer, un outil dédié à l'exécution d'Applets, livré avec le JDK.
- Inconvénient : ne permet de voir que l'Applet, pas la page HTML qui va autours.

```
> appletviewer HelloWorld.html
```

Les méthodes de la classe Applet

Le tag <applet>

- **Passage de paramètres à l'Applet**
  - Dans la page HTML

```
<applet ...liste de paramètres...>

<param name=nbPacman value="4">
<param name=imgFond value="toto.gif">

</applet>
```

- **Récupération des paramètres**
  - Dans le fichier java

```
...

int nbPacman;
public void init() {
    String param1 = getParameter("nbPacman");
    if(param1 == null) {
        nbPacman = 0;
    }
    else {
        try {
            nbPacman = Integer.parseInt("param1");
        catch(NumberFormatException e) {}
    }
}
...
```

Les méthodes de la classe Applet

Le tag <applet>

- **Syntaxe du tag <applet>**

```
<applet ...liste de paramètres...>

Texte au cas ou le navigateur ne com-
prend pas java

</applet>
```

- **Paramètres du tag <applet>**
  - code=”Test.class”
    - Nom de la classe principale.
  - width=200, height=300, taille en pixels
  - hspace=10, vspace=10, en pixels
    - Espace en pixel au dessus et autours de l’Applet
  - codebase=”codebaseURL”
    - URL de base de l’applet. Par exemple “java/classes”. Par défaut, le répertoire courant.
  - name=”Test_java”
    - permet de nommer une Applet. Plusieurs Applets nommées peuvent communiquer entre elles au sein d’une même page HTML.
  - align=alignement
    - valeurs possibles : left, right, top, texttop, middle, absmiddle, baseline, bottom, absbottom
  - archive=”test.jar” (java 1.1)
    - Nom de l’archive contenant l’ensemble des classes java de l’Applet.

Applications réseau

Programmation réseau

Avec Java, c'est facile !

- **Ouvrir des connexions par sockets.**
- **Créer un code serveur sous TCP/IP.**
- **Créer un code client sous TCP/IP.**
- **Idem sous UDP.**

Programmation réseau

Les sockets

- **Qu’est-ce qu’un socket ?**
  - Point d’entrée entre deux applications du réseau.
  - Pour les applications : descripteurs de fichiers.
  - Echange des données à l’aide des mécanismes d’entrées/sorties déjà étudiés.

- **Différent types de sockets**

1. Stream Socket (TCP)
   - Permettent d’établir une communication en mode connecté. Si la connexion est interrompue, les applications en sont informées.
2. Datagram Socket (UDP)
   - Communication en mode non connecté (Datagramme). Les données sont envoyées sous la forme de paquets indépendants de toute connexion. Plus rapide mais moins fiable que le mode TCP : aucune garantie que les paquets sont distribués correctement.
3. Raw Sockets
   - Permettent un accès direct aux couches les plus basses de la programmation réseau. Très bas niveau. Utilisés pour le debugging de protocoles. Non implémentés en Java.

Programmation réseau

Les “domaines” de socket

- **Domaine de socket ?**
  - Les sockets ont été créés pour être aussi génériques que possible en terme de communication inter-processus.
  - Organisés dans deux domaines permettant de définir entre eux la manière de communiquer : Unix et Inet (Internet).
  - Ces deux domaines permettent de définir si la communication est locale ou si elle opère sur le réseau entre deux systèmes différents.

- **Le domaine Unix**
  - Communication inter-processus locale à une même machine. Les deux modes de sockets (Stream et Datagram) sont supportés. Pas de Raw sockets.

- **Le domaine Internet**
  - Communication à travers le réseau,
  - tous modes de sockets supportés,
  - une fois la connexion établie, les données peuvent être lues ou écrites comme dans un fichier.

Programmation réseau

Le domaine Internet

- **Applications TCP les plus connues**
  - FTP, SMTP, TELNET, etc...
- **Applications UDP les plus connues**
  - Simple Network Management Protocol (SNMP).
  - Trivial File Transfer Protocol (TFTP), version Datagramme de FTPD, utilisée pour le boot par le réseau.
- **Les Raw Sockets permettent d'accéder à ICMP (Internet Control Message Protocol) et sont utilisés pour des manipulations de bas niveau.**
  - La commande “ping” utilise ICMP...

Programmation réseau
TCP/UDP

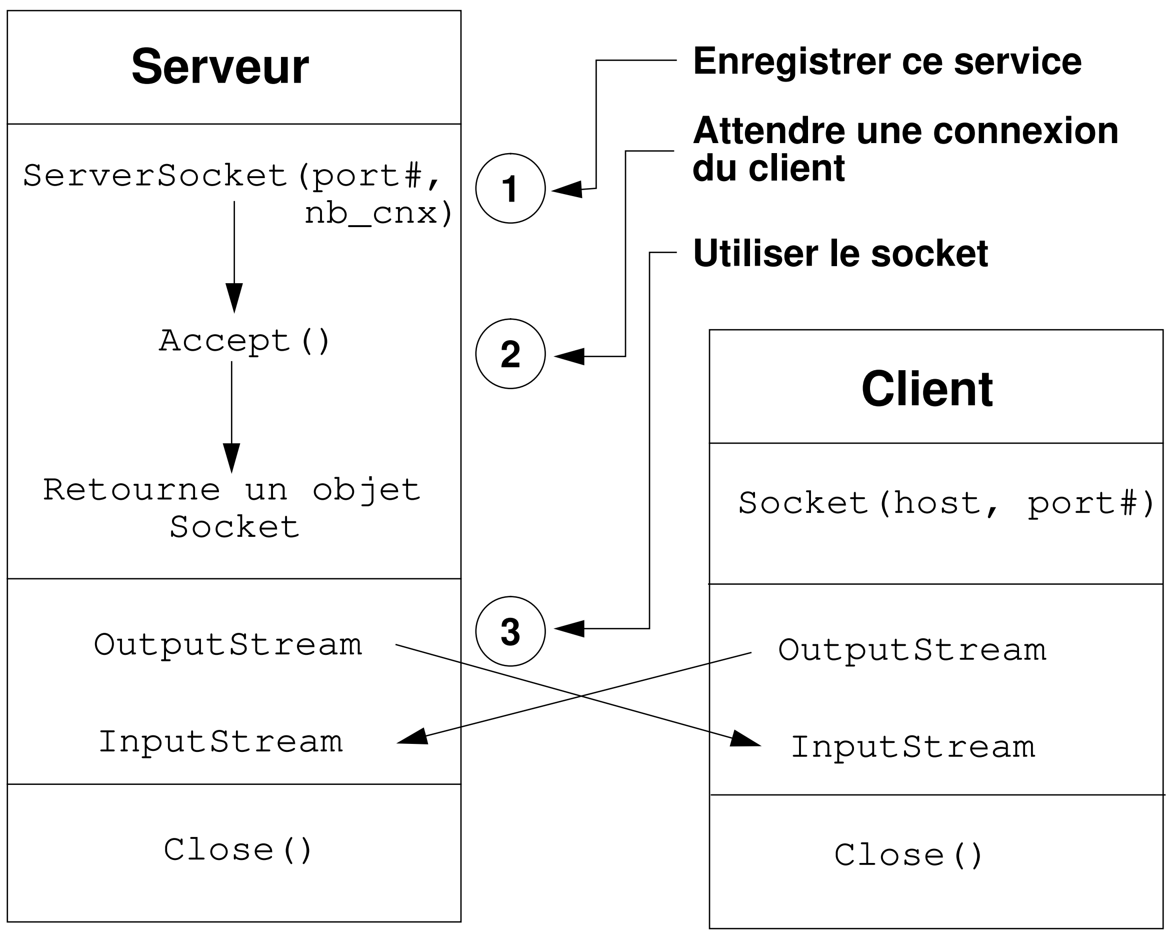
- En java, on ne travaille pas au niveau TCP ou UDP, on travaille au niveau de l'application, à l'aide des classes du package `java.net`.
- Les classes `URL`, `URLConnection`, `Socket`, et `SocketServer` utilisent TCP.
- Les classes `DatagramPacket` et `DatagramServer` utilisent UDP.
- Il n'est pas possible aujourd'hui de travailler en ICMP en Java.
- Certaines classes du package non documenté `java.sun` permettent de faire du ftp.

Programmation réseau

Le modèle réseau Java

- **Les sockets TCP/IP sont implémentés au travers des classes du package `java.net`.**

- **Principe de fonctionnement d'un modèle client-serveur :**



Programmation réseau

Le modèle réseau Java

1. Le serveur enregistre son service sous un numéro de port, indiquant le nombre de clients qu'il accepte de faire bufferiser à un instant T,

2. il se met en attente par la méthode `accept()` de son instance de `ServerSocket`,

3. le client peut alors établir une connexion en demandant la création d'un socket à destination du serveur pour le port sur lequel le service a été enregistré,

4. le serveur sort de son `accept()` et récupère un `Socket` de communication avec le client,

5. ils peuvent alors utiliser des `InputStream` et `OutputStream` pour échanger des données.

Programmation réseau

Principe d'un serveur TCP/IP

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

class simpleServer {
    public static void main(String args[]) {
    ServerSocket s = (ServerSocket) null;
    Socket s1;
    String sendString = "Message du serveur\n";
    int slength;
    OutputStream s1out;

    // Enregistrer le service sur le port 5432,
    // Autoriser 5 demandes simultanées.
    try {
        s = new ServerSocket(5432, 5);
    } catch(IOException e) {}

    // reboucler sur l'acceptation de toute
    //connexion entrante
    while(true) {
        try {
            // Attendre la connexion du client
            s1 = s.accept();

            // Extraire un stream du socket
            s1out = s1.getOutputStream();

            // Envoyer les données
            slength = sendString.length();
            for(i = 0; i < slength; i++)
                s1out.write((int)sendString.charAt(i));

            // Fermer la connexion (Socket), mais pas le
            // service (ServerSocket)
            s1.close();
        } catch(IOException e) {}
    }
}
```

Programmation réseau

Principe d'un client TCP/IP

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

class simpleClient {
   public static void main(String args[])
                              throws IOException {
       int c;
       Socket s;
       InputStream sIn;

       // Se connecter sur la machine "serveur",
       // port 5432
       s = new Socket("serveur", 5432);

       // Acquerir un InputStream, lire les infos
       // en provenance du serveur
       sIn = s.getInputStream();
       while((c = sIn.read()) != -1) {
           System.out.println((char) c);

       // Lorsque EOF est atteint, fermer la
       // connexion
       s.close();
   }
}
```


Programmation réseau

Remarques

- Je vous recommande d'utiliser des `DataInputStream` et des `DataOutputStream` pour réaliser l'échange de données

```
...
Socket s = new Socket("www.essi.fr", 13);
DataInputStream dis =
   new DataInputStream(s.getInputStream());
String line;

while((line = dis.readline()) != null)
   System.out.println(line);
...
```

- L'exemple précédent accepte plusieurs connexions simultanées, mais ne traite qu'un client à la fois, les autres sont mis en attente
- Corriger cette limitation : utiliser les threads !

```
while (true) {
   accepter une connexion ;
   Créer un thread qui va s'occuper d'échanger les
   données avec le client
}
```

Programmation réseau
Utilisation des sockets Datagrams

- **Utiliser les classes `DatagramPacket` et `Data-gramSocket`.**

- **Ces objets sont initialisés différemment selon qu'ils sont utiliser pour envoyer ou recevoir des paquets.**

- **Envoi d'un Datagram :**

  1. Créer un **`DatagramPacket`**, en spécifiant les données à envoyer, leur longueur, la machine réceptrice et le port,
  2. utiliser la méthode **`sent(DatagramPaquet)`** de la classe **`DatagramSocket`**. *Le constructeur de ce dernier ne prend pas d'arguments*, toutes les informations se trouvent dans le paquet envoyé.

Programmation réseau
Envoi d'un Datagram

```
...
// Machine destinataire
InetAddress address =
       InetAddress.getByName("jessica.essi.fr");
static final int port=4444;

// Création du message à envoyer
String s = new String("coucou");
int longueur = s.length();
byte[] message = new byte[longueur];
s.getBytes(0, longueur, message, 0);

// Initialisation du paquet avec toutes les infos
DatagramPacket p = new DatagramPacket(message,
                       longueur, address, port)

// Création du socket et envoi du paquet
DatagramSocket s = new DatagramSocket();
s.send(p);
...
```

Programmation réseau

Réception d'un Datagram

- **Pour recevoir un Datagram**

1. Créer un **`DatagramSocket`** qui écoute sur le bon port du host destinataire,
2. créer un **`DatagramPacket`** avec un buffer suffisamment grand pour recevoir les paquets envoyés par le serveur,
3. utiliser la méthode **`receive()`** (bloquante) de la classe **`DatagramPaquet.`**

```
...
// Buffer de réception
byte[] buffer = new byte[1024];
String s;
// Paquet associé à un buffer vide pour réception
DatagramPacket p = new DatagramPacket(buffer, buf-
fer.length);
// Socket qui écoute sur le port
DatagramSocket s = new datagramSocket(port);
for(;;) {
    // Attente de reception
    socket.receive(p);
    // Affichage du paquet récu
    s = new String(buffer, 0, 0, p.getlength());
    System.out.println(“Paquet reçu de “ +
                    p.getAddress().getHostName() +
                    “ : “ + p.getPort() + “ : “ + s);
```

Un exemple tiré du tutorial de Sun
Implémentation d'un protocole en UDP

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

class QuoteServerThread extends Thread {
    private DatagramSocket socket = null;
    private DataInputStream qfs = null;

    QuoteServerThread() {
        super("QuoteServer");
        try {
            socket = new DatagramSocket();
            System.out.println("QuoteServer listening on
                        port: " + socket.getLocalPort());
        } catch (java.io.IOException e) {
            System.err.println("Could not create datagram
                                socket.");
        }
        this.openInputFile();
    }
...
```


Un exemple tiré du tutorial de Sun
Implémentation d'un protocole en UDP

```
public void run() {
    if (socket == null)
        return;

    while (true) {
        try {
            byte[] buf = new byte[256];
            DatagramPacket packet;
            InetAddress address;
            int port;
            String dString = null;

            // receive request
            packet = new DatagramPacket(buf, 256);
            socket.receive(packet);
            address = packet.getAddress();
            port = packet.getPort();

            // send response
            if (qfs == null)
                dString = new Date().toString();
            else
                dString = getNextQuote();

            dString.getBytes(0, dString.length(), buf, 0);
            packet = new DatagramPacket(buf, buf.length,
                                        address, port);
            socket.send(packet);
        } catch (IOException e) {
                System.err.println("IOException:  " + e);
                e.printStackTrace();
            }
        }
    }
    ...
```

Un exemple tiré du tutorial de Sun

Implémentation d'un protocole en UDP

```
protected void finalize() {
    if (socket != null) {
        socket.close();
        socket = null;
        System.out.println("Closing datagram socket.");
    }
}

private void openInputFile() {
    try {
        qfs = new DataInputStream(new
                    FileInputStream("one-liners.txt"));
    } catch (java.io.FileNotFoundException e) {
        System.err.println("Could not open quote file.
                            Serving time instead.");
    }
}

private String getNextQuote() {
    String returnValue = null;
        try {
            if ((returnValue = qfs.readLine()) == null) {
                qfs.close();
                this.openInputFile();

                // we know the file has at least one
                // input line!
                returnValue = qfs.readLine();
            }
        } catch (IOException e) {
            returnValue = "IOException occurred in
                            server.";
            }
        return returnValue;
    }
}
```

Un exemple tiré du tutorial de Sun
Implémentation d'un protocole en UDP

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

class QuoteClient {

  public static void main(String[] args) {
      int port;
      InetAddress address;
      DatagramSocket socket = null;
      DatagramPacket packet;
      byte[] sendBuf = new byte[256];

      if (args.length != 2) {
          System.out.println("Usage: java QuoteClient <hostname>
          <port#>");
          return;
      }
      try {
          // bind to the socket
          socket = new DatagramSocket();
      } catch (java.io.IOException e) {
          System.err.println("Could not create datagram
                              socket.");
      }

      if (socket != null) {
          try {
              // send request
              port = Integer.parseInt(args[1]);
              address = InetAddress.getByName(args[0]);
              packet = new DatagramPacket(sendBuf, 256,
                                          address, port);
              socket.send(packet);
              ...
```

Un exemple tiré du tutorial de Sun

Implémentation d'un protocole en UDP

```
            ...
            // get response
            packet = new DatagramPacket(sendBuf, 256);
            socket.receive(packet);
            String received = new String(packet.getData(), 0);
            System.out.println("Quote of the Moment: " +
                                                    received);
            socket.close();

        } catch (IOException e) {
            System.err.println("IOException:  " + e);
            e.printStackTrace();
          }
      }
  }
}
```

Le multithreading

Threads et MultiThreading

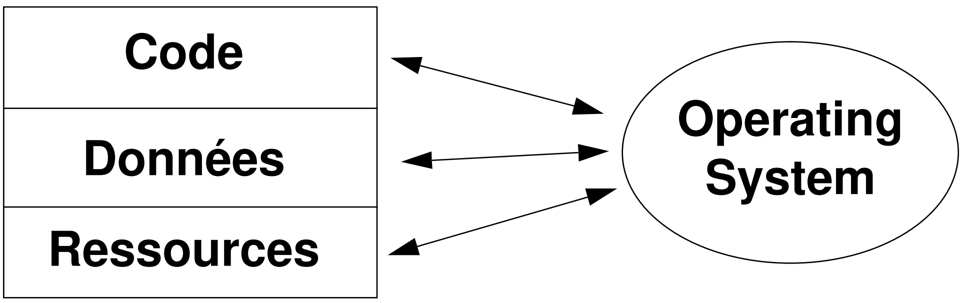
Objectifs

- **Définir un thread**
- **Comprendre l'utilité du multithreading**
- **Connaître les deux types de threads utilisables en Java**
- **Définir les différents états d'un thread et les méthodes permettant de modifier ces états**
- **Déterminer la priorité d'un thread**
- **Connaître les différentes possibilités de synchronisation de thread**

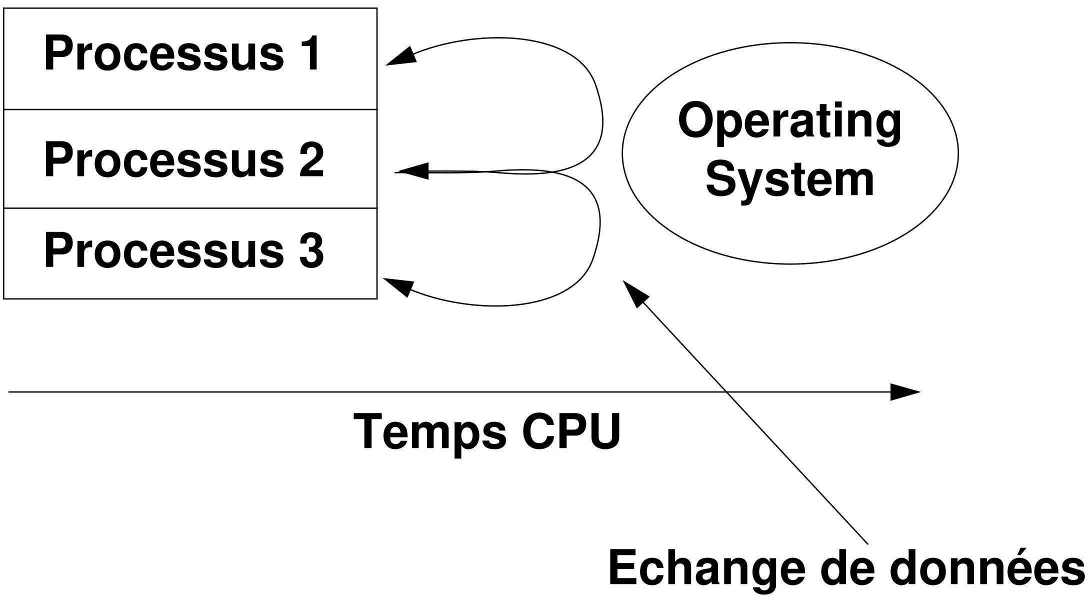
Threads et MultiThreading

Programmation "parallèle"

- **Processus = unité fonctionnelle**



- **Exécution**

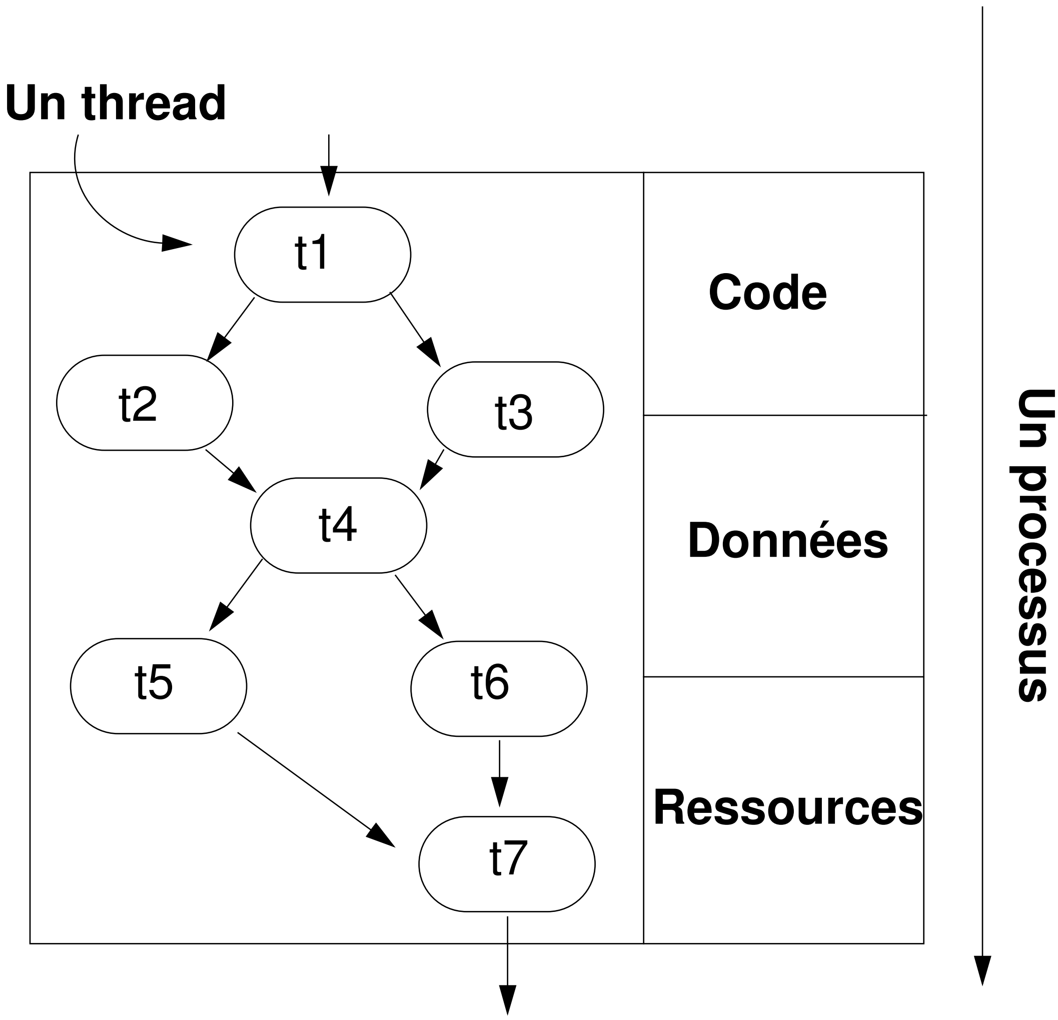


- **Inconvénients d'un processus**
  - Coûteux en mémoire (code + données non partagés),
  - lancement lent (taille, charge de l'OS),
  - partage des ressources systèmes difficile...

Threads et MultiThreading

Qu’est-ce qu’un Thread ?

- **Processus “interne” à un processus**
- **Un thread partage les données, le code et les ressources de son processus**
- **Il peut disposer de ses données propres**



Threads et MultiThreading

Qu’est-ce qu’un Thread ?

- **Un processus comporte plusieurs threads**
- **Les threads partagent code, données, ressources, mais peuvent disposer de leurs propres données**
- **Ils peuvent s’exécuter en parallèle (OS dépendant...)**
- **Avantages des threads sur les processus**
  - Légèreté (“light weight process” ou “processus léger”) grâce au partage des données.
  - Ceci implique des meilleures performances au lancement et en exécution.
  - Partage des ressources système du processus (très pratique, ne serait-ce qu’au niveau des entrées/sorties).
- **Utilité**
  - Puissance de la modélisation : le monde est “multithread” !
  - Puissance d’exécution grâce au paraléllisme.
  - Simplicité d’utilisation : un Thread est un Objet Java : attributs, méthodes, encapsulation, réutilisation...

Threads et MultiThreading

Un exemple

- **Deux vérificateurs d'une usine doivent contôler le contenu de sacs d'objets**

```
public class Thread1 {
   public static void main(String args[]) {
       Sac plus = new Sac(10, "+");
       Sac moins = new Sac(10, "-");
       Verificateur Gaston =
           new Verificateur("gaston", plus);
       Verificateur Robert =
           new Verificateur("robert", moins);

       // début du contrôle
       Gaston.run();
       Robert.run();
   }
}
```

```
class Sac {
   String contenu;
   int taille;
   public Sac(int taille, String contenu) {
       this.taille = taille;
       this.contenu = contenu;
   }
}
```

Threads et MultiThreading

Un exemple (suite)

```
class Verificateur {
    private Sac sac;
    private String nom;

    public Verificateur(String nom, Sac sac) {
        this.nom = nom;
        this.sac = sac;
    }

    public void run() {
        for(int i = 0; i < sac.taille; i++) {
            System.out.print(sac.contenu);
        }
        System.out.printl("\n" + nom + "a terminé");
    }
}
```

- **Résultat :**

```
++++++++++
Gaston a terminé
----------
Robert a terminé
```

  - C’est séquentiel ! En fait on a pas utilisé de thread ici !

Threads et MultiThreading

Mais alors, dites-nous comment faire !

- **Première approche : utiliser la classe Thread**
  - java.lang.Thread est la classe d'implémentation d'un thread
    - création, destruction,
    - gestion de l'état,
    - synchronisation,
    - etc...
  - Le code d'une instance de Thread s'exécute parallèlement au code des threads déjà existants

- **Utilisation**
  - Sous-classer Thread
    - Spécialiser une classe existante et enrichir son comportement.
  - Associer un thread à un objet
    - Définir une aggrégation ou association entre la classe à utiliser et la classe thread.

Threads et MultiThreading

Solution 1 : sous-classer Thread

```
class Verificateur extends Thread {
    private Sac sac;
    private String nom;

    public Verificateur(String nom, Sac sac) {
        this.nom = nom;
        this.sac = sac;
    }

    public void run() {
        for(int i = 0; i < sac.taille; i++) {
            System.out.println(sac.contenu);
            try {
                sleep(100);
            } catch(Exception e) {}
        }
        System.out.printl("\n" + nom + "a terminé");
    }
}
```

Threads et MultiThreading

Solution 1 : sous-classer Thread

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

        Sac plus = new Sac(10, "+");

        Sac moins = new Sac(10, "-");

        Verificateur Gaston =
            new Verificateur("gaston", plus);

        Verificateur Robert =
            new Verificateur("robert", moins);


        // début du contrôle

        // Utilisation de la méthode start()
        // héritée par la classe Verificateur.
        // start() appelle run()

        Gaston.start();

        Robert.start();

    }
```

- **`start()`** rend la main de suite après son appel

- **Résultat**

```
+-+-+-+-+-+-+-+-+-+-

Gastom a terminé

Robert a terminé
```

Threads et MultiThreading

Solution 2 : implémenter l'interface runnable

- **Un thread peut partager le contexte d'un objet existant, qui à l'origine n'est pas un Thread.**
- **Cet objet doit implémenter l'interface Runnable.**
- **L'exemple suivant montre également comment associer plusieurs Threads à un seul objet.**

Threads et MultiThreading

Solution 2 : un exemple

```
class Verificateurs extends toto implements
                                          Runnable {
    private SacVector sacVect;
    private int IndexSacCourant = 0;

    public void gereNouveauSac(Sac sacAGerer) {
        Thread unVerificateur = new Thread(this);
        sacVect.add(sacAGerer, unVerificateur);
        unVerificateur.start();
    }

    // implémentation de Runnable
    public void run() {
        sac monSacAVerifier;
        Thread moi = Thread.getCurrentThread();
        monSacAVerifier = sacVect.chercheQui(moi);

        for(int i = 0; i < sac.taille; i++)
            System.out.println(monSac.contenu);
        System.out.printl("\n" + nom + "a terminé");
        sacVect.checkFiniPour(monSac);
    }
}
```

- Les éléments en italiques sont supposés définis dans la classe SacVector.
    - **`sacVect.add(sacAGerer, unVerificateur)`** associe un sac avec un thread,
    - **`sacVect.chercheQui(moi)`** retourne le sac corespondant au Thread passé en paramètre,
    - **`sacVect.checkFiniPour(monSac)`** gère le fait que le sac ait été contrôlé. Par exemple : l'ôte du vecteur et le transmet.

- **On utilise le constructeur de code en passant en paramètre l'objet dont on veut paralléliser le code (this).**

- **Ce paramètre doit implémenter `Runnable`, c.a.d que l'objet courant doit implémenter la méthode `run()`.**

- **Lorsque le thread est lancé par `start()` une entité exécutable supplémentaire est lancée pour l'objet, utilisant son code et ses attributs, ayant pour point d'entrée la méthode `run()`.**

- **A un instant t, un nombre quelconque de threads peuvent être associés à une instance de Verificateur. Ils partagent tous le même `SacVector`.**

- **Méthodes essentielles liées à cette technique**
  - **`public Thread(Runnable)`**
  - **`public static Thread currentThread()`**

- **Attention : lorsque l'on est dans le code de la méthode `run()`, this retourne une référence sur le Verificateur, quelque soit le Thread qui exécute cette méthode. Si on veut récupérer une référence sur le Thread lui-même, il faut utiliser `Thread.currentThread()`.**

- **La pile d'un Thread étant locale, toute variable créée dans `run()` est unique par Thread, en revanche, tous les attributs de la classe sont partagés par les threads.**

Threads et MultiThreading

Quelle solution choisir ?

- **Sous-classer Thread**
  - Lorsqu'on désire paralélliser une classe qui n'hérite pas déjà d'une autre classe

- **Implémenter Runnable**
  - Lorsque la super classe est imposée (exemple : une applet que l'on désire paralélliser),
  - Ou lorsqu'un partage de contexte est requis par un nombre quelconque de threads (cf l'exemple des verificateurs).
  - En général, le démarrage du ou des threads associés à l'objet est effectué par l'objet lui-même

```
(new Thread(this)).start();
```

Threads et MultiThreading

Les états d'un Thread

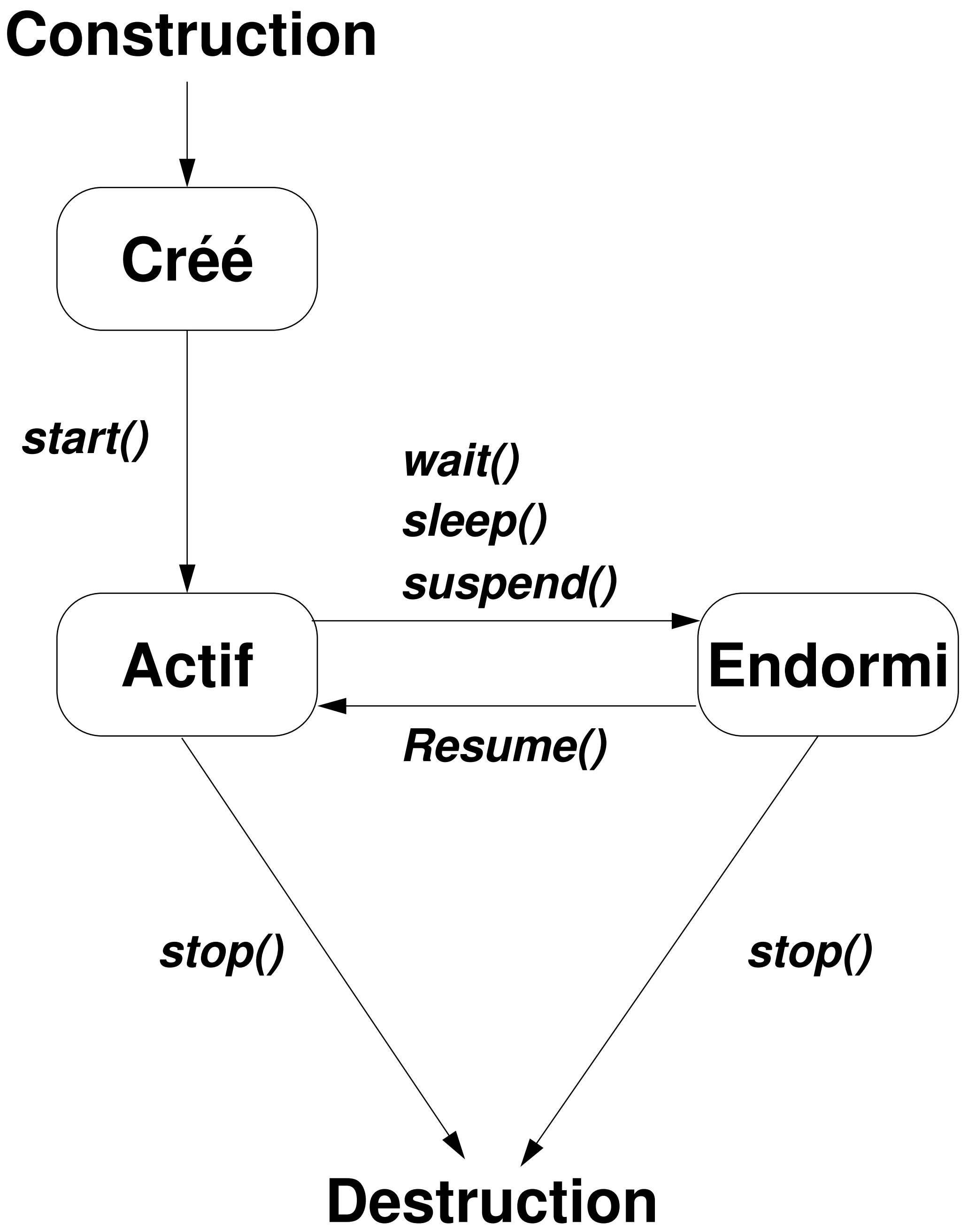


Table 1: isAlive()

	créé	**actif**	**endormi**	**mort**
isAlive()	false	true	true	false

Threads et MultiThreading

Les états d'un Thread

- **Créé**
  - Créé comme n'importe quel objet Java.
- **Actif**
  - Dans cet état lorsque après la création, il est activé par **`start()`**, qui lance **`run()`**.
  - Ou encore lorsqu'il est réactivé par un resume().
  - Il est alors ajouté à la liste des threads actifs qui sont exécutés en temps partagés.
- **Endormi**
  - **`sleep(long millisecondes)`** endort le thread pendant l'intervalle de temps spécifié.
  - **`suspend()`** endort un thread et **`resume()`** pourra le réactiver.
  - Une entrée/sortie bloquante (ouverture de fichier, entrée clavier) endort et réveille un thread.
- **Mort**
  - Quand **`run()`** a terminé son exécution.
  - Par un appel explicite à **`stop()`**.

Threads et MultiThreading
Priorités

- **Principes**
  - Java offre la possibilité de fixer les priorités des différents threads d'un processus.
  - Niveaux de priorités absolus (gestion simplifiée).
  - Seuls les threads à l'état actif s'exécutent et peuvent partager le CPU.
  - Le partage de temps entre threads d'une même priorité dépend de l'OS.
  - Les threads actifs de plus hautes priorité se partagent le CPU (si l'OS le permet).

- **Méthodes**
  - **`void setPriority(int)`** modifie la priorité du receveur. Le paramètre appartient à l'intevalle **`[MIN_PRIORITY, MAX_PRIORITY]`**. Dans le cas contraire **`IllegalArgumentException`** est levée.
  - **`int getPriority()`** permet de connaître la priorité d'un thread.
  - Le niveau de priorité “normal” est donné par la constante **`NORM_PRIORITY`**.

Threads et MultiThreading

Priorités

- **Partage de la ressource CPU. Deux types de gestion existent dans les OS du marché.**

- **Time-slicing : la ressource CPU est partagée équitablement entre threads de même priorité.**
  - Le système répartit équitablement le CPU entre processus de même priorité. Ils s'éxécutent en parallèle...

- **Préemption : le partage du CPU doit être géré par l'utilisateur.**
  - Le premier thread du groupe des threads à priorité égale monopolise le CPU. Il faut qu'il le cède soit involontairement, soit volontairement, pour que les autres threads puissent s'exécuter.
  - Involontairement sur Entrée/sortie,
  - implicitement sur passage à l'état endormi par **`wait(), sleep()`** ou **`suspend(),`**
  - volontairement par appel à la méthode statique **`yield()`**. Cette méthode ne permet cependant pas à un thread de priorité inférieure de s'exécuter. *Seul un thread de priorité égale peut prendre la main*.

Threads et MultiThreading

Synchronisation

- **Certains accés simultanés à une ressource peuvent être gênants.**
- **Gérer les concurrences d'accés à une méthode**
  - Cela se fait en déclarant la méthode **`synchronized`** :

```
public synchronized void maMethode() {
    ...
}
```

  - Lorsqu'un thread exécute cette méthode sur un objet, un autre thread ne peut pas l'exécuter pour le même objet.
  - En revanche, il peut exécuter cette méthode pour un autre objet.

Threads et MultiThreading

Synchronisation

- **Contôler l'accés à un objet**
  - Déclarer l'objet synchronized

```
public void maMethode() {
    ...
    synchronized(objet) {
        objet.methode();
    }
    ...
}
```

  - L'accès à l'objet passé en paramètre de **`synchronized(Object obj)`** est réservé à un thread et un seul.

- **Pour éviter une dégradation des performances il faut que les sections critiques soient courtes et utilisées à bon escient.**

Threads et MultiThreading

Daemons

- **Exemple de daemons : les threads de service : gargage collector, afficheur d'image, etc...**
- **En général ils ont une faible priorité, et tournent au sein d'une boucle infinie.**
- **Arrêt implicite lorsque le programme se termine.**
- **Méthodes**
    - **`setDaemon()`** : permet de déclarer un thread comme un daemon
    - **`isDaemon()`** : permet de savoir si un thread est un daemon
- **Les daemons ne sont pas étudiés plus en détail dans ce cours...**

Threads et MultiThreading
Groupes de threads

- **Groupe de threads par défaut : main**

- **Arborescence de Threads et de ThreadGroups**
  - La classe **`ThreadGroup`** permet de constituer une arborescence de Threads et de ThreadGroups
  - Elle offre des méthodes de manipulation récursives d'un ensemble de threads telles que **`resume(), stop(), suspend()...`**
  - Comporte des méthodes qui agissent seulement sur les Threads créés ensuite dasn un groupe : **`setDaemon(), setMaxPriority()...`**

- **Fonctionnement**
  - La machine virtuelle crée au minimum un groupe de threads appelé main.
  - Par défaut tout thread appartient au même groupe que son père.
  - Possibilté de spécifier un ThreadGroup à la construction d'un Thread, main si on ne précise pas.
  - Tout thread est capable de connaître son threadGroup par **`getThreadGroup()`**.

Threads et MultiThreading

Groupes de threads

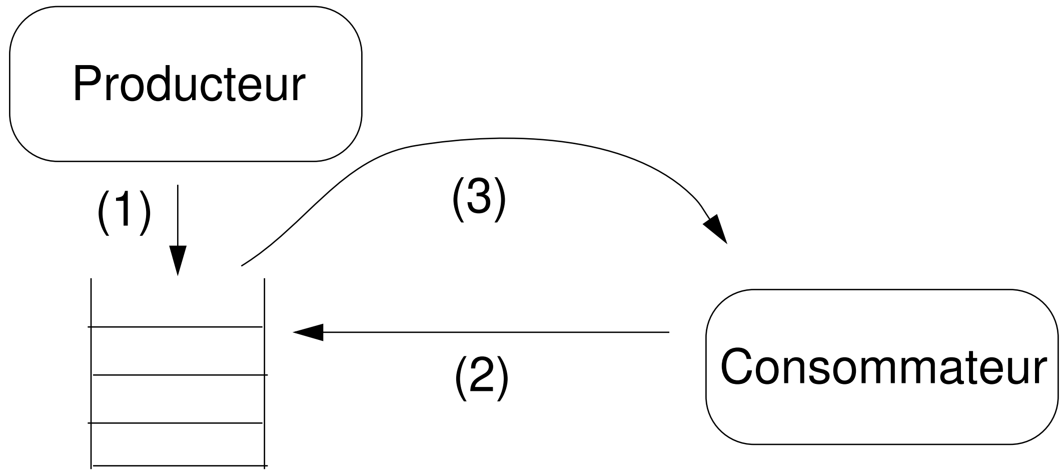
```
public static void main(String args[]) throws
                          InterruptedException {

   Verificateur Gaston =
      new Verificateur("gaston",
                              new Sac(10,"+"));

   Verificateur Robert =
      new Verificateur("robert",
                          new Sac(10,"-"));

   // Création du groupe des vérificateurs

   ThreadGroup lesVerificateurs =
         new ThreadGroup("Verificateurs");

   Thread t1 = new Thread(lesVerificateurs,
                                 Gaston);

   Thread t2 = new Thread(lesVerificateurs,
                                 Robert);

   // Lancements individuels

   t1.start();

   t2.start();

   ...

   // Opérations sur le groupe

   lesVerificateurs.suspend();

   ...

   lesVerificateurs.resume();

}
```


Threads et MultiThreading

Rendez-vous

- **Exemple producteur/consommateur**

  1. Le producteur empile les données dans le buffer.
  2. Le consommateur demande au buffer une information.
  3. Lorsqu'une information est disponible, il la prend.



- **Le producteur et le consommateur sont deux threads qui s'exécutent en parallèle.**

- **Ils prossèdent chacun leur propre rythme de production et de consommation.**

- **Si le buffer est vide, le consommateur est mis en attente jusqu'à ce que le producteur dépose quelque chose.**

Threads et MultiThreading

Rendez-vous

```
class Producteur extends Thread {
    private Buffer buffer;
    private String donnee;

    public producteur(Buffer buffer,
                                String donnee){
        this.buffer = buffer;
        this.donnee = donnee;
    }

    public void run() {
        for(int i = 0; i < donnee.length; i++) {
            // Dépôt des données
            buffer.poser(donne.charAt(i));
            try {
                // rythme de production aléatoire
                sleep((int) (Math.random() * 25));
            } catch(Exception e) {}
        }
        System.out.println("Production terminée");
    }
```

- `Run()` **dépose chaque caractère de la chaîne donnee en activant la méthode** `poser()` **du buffer**

Threads et MultiThreading

Rendez-vous

```
class Buffer extends java.util.Stack {
    public synchronized void poser(char donnee) {
        // Attente tant que le buffer est plein
        while(full()) {
            try {
                // Mise en attente
                wait();
            } catch (Exception e) {//traite e}
        }

        // Il y a une place libre
        push(new Character(donnee));
        // Fin de la mise en attente
        notify();
    }
```

- **La classe `Buffer` étend java.util.Stack qui implémente une pile First In Last Out (FILO)**
- **Si le buffer est plein, la méthode `poser()` :**
    - Met le thread appelant en attente avec **`wait()`**,
    - Le libère avec **`notify()`** quand un espace est diponible, c'est-à dire quand le consommateur est venu prendre un élément.

Threads et MultiThreading

Rendez-vous

```
class Consommateur extends Thread {
    private Buffer buffer;
    private int nb;

    public consommateur(Buffer buffer, int nb) {
        this.buffer = buffer;
        this.nb = nb;
    }

    public void run() {
        // boucle de lecture des données
        for(int i = 0; i < nb; i++) {
            char car = buffer.prendre();
            System.out.print(car);
            try {
                // rythme de consommation aléatoire
                sleep((int) (Math.random() * 500));
            catch(Exception e) {}
        }
        System.out.println("\n terminée");
    }
}
```

- La méthode **`run()`** du consommateur effectue une itération pour prendre des caractères dans le buffer.

Threads et MultiThreading

Rendez-vous

```
class Buffer extends java.util.Stack {
    public synchronized char prendre() {
        // Attente tant que le buffer est vide
        while(empty()) {
            try {
                // Mise en attente
                wait();
            } catch (Exception e) {}
        }
        // Libération du buffer
        notify();
        // Retour de la donnée
        return((Character) pop()).charValue();
    }
```

- **Si le buffer est vide, le buffer met le consommateur (thread appelant) en attente jusqu'à ce que le producteur pose un nouvel élément. Il le libère alors par `notify()` .**

Threads et MultiThreading

Rendez-vous

- **Programme principal**

```
public class ProdCons {

   public static void main(String argv[]) {

      String donnee = "Rendez-vous avec java";
      Buffer buffer = new Buffer();
      Producteur producteur =
                new Producteur(buffer, donnee);
      Consommateur conso =
         new Consommateur(buffer,
                            donnee.length());

      producteur.start();
      conso.start();

   }

}
```

- **Résultats :**

```
Rndez-vous avec Jav

production terminée

ae

Consommation terminée
```

- **Le “n” de “rendez” a été produit avant que le “e” ne soit consommé.**

Threads et MultiThreading

Rendez-vous

- **La classe Buffer**

```
class Buffer extends java.util.Stack {

   public synchronized void poser(char donnee) {

       // Attente tant que le buffer est plein
       while(full) {
           try {
               wait(); // mise en attente
           } catch(Exception e) {}
       }

       // il y a une place libre
       push(new Character(donnee));

       notify(); // fin de mise attente

   }


   public synchronized char prendre() {

       // attente tant que le buffer est vide
       while(empty);
           try {
               wait(); // mise en attente
           } catch(Exception e) {// traite e}
       }

       notify(); // libération

       // retour de la donnée
       return ((Character)pop()).charValue();
   }

   public boolean full() { return(size() == 2); }

   public boolean empty() { return(size == 0); }

}
```

Multimédia en java, images et sons

Afficher une image
Principe de base

```
import java.awt.*;
import java.applet.*;

public class monApplet extends Applet {
   private Image im;

   public void init() {
      im = getImage(getDocumentBase(),
                          "images/toto.gif");
   }

   public void paint(Graphics g) {
      g.drawImage(im, 25, 25, this);
}
```

- **getImage(...) ne charge pas l'image !**
  - Création de l'objet Image
- **g.drawImage(...)**
  - lance un thread de chargement de l'image la première fois qu'elle est invoquée
  - Génère des repaint() tant que l'image n'est pas arrivée. Pas super...

La classe Mediatracker

- **Contrôle sur le chargement asynchrone des images**

- **Offre des méthodes permettant de savoir si une ou plusieurs images sont totalement chargées, s'il n'y a pas eu d'erreurs, etc...**

Composant sur lequel se fait l'affichage de l'image

```
public void init() {
    images = new Image[10];
    MT = new Mediatracker(this);
    for(int i = 0; i < 10; i++) {
        images[i] = getImage(getDocumentBase(),
                        "images/im" + i + ".gif");
        MT.addImage(images[i], i);
    }
    try {
        MT.waitForAll();
    } catch(InterruptedException e) {}
    // phase de chargement terminée, attention
    // les images ne sont pas forcément arrivées...
    ...
}
public void paint (Graphics g) {
    if(MT.statusID(index, false) &&
                        MediaTracker.COMPLETE) {
        g.drawImage(images[index]);
    }
```

Identificateur pour le tracking

La classe Mediatracker

Méthodes utiles

- **boolean checkAll()**
  - renvoie true si toutes les images sont chargées en mémoire
- **int statusId(int id, boolean to_reload)**
  - Renvoie l'état du chargement de l'image i
- **boolean checkId(int id)**
  - Renvoie true so l'image i est chargée en mémoire
- **boolean waitForId(int id)**
  - Attend que l'image i soit chargée
- **boolean isErrorId(int id)**
  - Renvoie true si une erreur est intervenue lors du chargement de l'image i

PixelGrabber et MemoryImageSource
Manipulation avancée d'images

- **PixelGrabber permet de produire un tableau d'entiers à partir d'une Image**
- **MemoryImageSource réalise l'opération inverse**
- **Utiles pour la synthèse d'image ou pour faire des effets spéciaux (dissolve, fading, etc...)**

```
im1 = getImage(getDocumentBase(), "toto.gif");
int w = im1.getWidth(this);
int h = im1.getHeight(this);
int pixels[] = new int[w *h];
PixelGrabbler pg;
   pg = new PixelGrabbler(im1, 0, 0, w, h,
                          pixels, 0, w)
   try {
      // remplissage du tableau de pixels
      pg.grabPixels();
   catch(InterruptedException e) {}
      // manipulation du tableau pixels
      eclaircit(pixels);
      // recopie dans l'image
      MemoryImageSource im =
         new MemoryImageSource(w, h, pixels,
                               0, w);
      im2 = createImage(mem);
```

Tables des matières