Programmation Orientée Objets

www.polytech.unice.fr/~vg

Granet Vincent – vg@unice.fr

Polytech – Elec5 – S9/2020

Sommaire

Bibliographie

Vincent Granet.
Algorithmique et programmation en Java.
Dunod, 5e édition, 2018.

Vincent Granet et Jean-Pierre Regourd.
Aide-Mémoire de Java.
Dunod, 5e édition, 2019.

http://users.polytech.unice.fr/~vg/index-peip2.html.

Introduction

Historique

- 1991, J. Gosling et Sun. Langage pour programmer des processeurs embarqués dans des appareils électroménagers
- Origine du nom : Oak puis Java (kawa)
- 1994, Java utilisé pour écrire un navigateur Web (futur HotJava)
- Netscape inclut un interprète Java dans son navigateur
- 2000, Java 2 (J2SE - J2EE)
- Java langage de programmation à usage général + application pour le Web (applet = little application)
- Java embarqué : PDA, téléphones mobiles
- 2009 Oracle rachète Sun
- 2011, Java 7
- 2014, Java 8 - λ-fonctions
- 2018, Java 9 - Modules - Java 10 - inférence de type

Avantages

- langage à objets, mais aussi multi-paradigmes
- syntaxe simple
- notation issue de C
- portabilité
- API très vaste
- javadoc
- interfaces graphiques (awt, Swing)
- environnement jdk gratuit

Inconvénients

- produit industriel
- temps d'exécution - Interprétation (mais JIT)
- API très (trop ?) vaste, difficile à maîtriser
- constructions archaïques issues de C
- temps réel

Premières applications

```java
/* ma première application Java */
class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }
} // fin classe HelloWorld
```

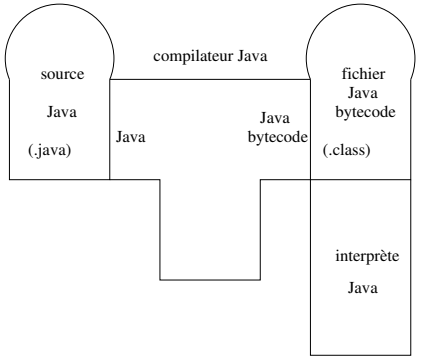
```java
/* ma première applet Java */
import java.awt.Graphics;
import java.applet.Applet;

public class HelloWorldApplet extends Applet {
    public void paint(Graphics g) {
      g.drawString("Hello World!", 5, 25);
    }
} // fin classe HelloWorldApplet
```

Compter les caractères

```
/** La classe Wc écrit le nombre de caractères
 *  lus sur l'entrée standard
 */
import java.io.*;
public class Wc {
    public static void main (String[] args)
        throws IOException
    {
        int nbc = 0; // compteur de caractères

        while (System.in.read() != -1)
              nbc++;
        // fin de fichier de l'entrée standard
        System.out.println(nbc);
    }
} // fin classe Wc
```


Mise en œuvre

- interprétation



- la variable `CLASSPATH`
- Production code à la volée (JIT)

Comment compiler et exécuter ?

```
$ javac  HelloWorld.java

$ java   HelloWorld
Hello World!

$ javac   HelloWorldApplet.java

$ appletviewer  HelloWorldApplet.html
```

```
<object  type="application/x-java-applet"  width=400 height=150>
  <param name="code" value="HelloWorldApplet.class">
</object>
```


La documentation

- javadoc
- produit une documentation en html
- traite les commentaires `/** */`
- reconnaît des macros :
  - @author
  - @version
  - @param
  - @return
  - @see
  - ...

```
$ javadoc Wc.java
$ firefox index.html
```

Types Élementaires

Les nombres

- Arithmétique classique
- **byte** (8 bits)
- **short** (16 bits)
- **int** (32 bits)
- **long** (64 bits)

```
123 0 98 067 0xAeF1
Byte.MIN_VALUE Byte.MAX_VALUE
Short.MIN_VALUE Short.MAX_VALUE
Integer.MIN_VALUE Integer.MAX_VALUE
Long.MIN_VALUE Long.MAX_VALUE
```

- **float** (32 bits)
- **double** (64 bits)

```
123.12 0. .12 98.  3.5e-7  1.5e2
Float.MIN_VALUE Float.MAX_VALUE
Double.MIN_VALUE Double.MAX_VALUE
```

Les booléens

- **`boolean`**
- **`false`** et **`true`**
- opérateurs :
  - `!` la négation
  - `|` la disjonction
  - `^` la disjonction exclusive
  - `&` la conjection
  - `||` la disjonction conditionnelle
  - `&&` la conjection conditionnelle

Les caractères

- **`char`**
- jeu de caractères Unicode
- `www.unicode.org`
- constantes dénotées en apostrophes (*e.g.* `'a'`, `'4'`, `' '`, etc.)
- caractères spéciaux :

  `\b` `\f` `\n` `\r` `\t` `\\` `\'`
- valeur hexadécimale : `\uxxxx` (*e.g.* `\u0041`)
- **`int`** `nbElèves;`
- **`double`** Δ`;`

Les conteneurs

- Les valeurs des types élémentaires ne sont pas des *objets* (au sens de la programmation objet)

```
Byte Short Integer Long
Float Double
Boolean
Character
```

- mais conversions implicites (depuis jdk1.5)

```
Integer i = 5;
char c = new Character('z');
```


Les énoncés

Les énoncés conditionnels

- même sémantique que C
- `if`
- `switch`

Les énoncés itératifs

- même sémantique que C, plus énoncé *foreach*
- `while`
- `do-while`
- `for`

L'énoncé *foreach* (1/2)

Forme généralisée de l'énoncé `pour` lorsqu'il s'agit d'appliquer un même traitement à tous les éléments d'un tableau ou d'une collection :

- écriture plus naturelle et plus sûre

```
int [] tab;
....
int somme = 0;
for (int i=0; i <tab.length; i++)
    somme += tab[i];
```

```
int [] tab;
....
int somme = 0;
for (int x : tab)
    somme+=x;
```


Programmation Orientée Objets
└Les Énoncés

L'énoncé *foreach* (1/2)

```
Vector<Ingeger> v = new Vector<Integer>();
v.addElement(13); v.addElement(128); ....
int somPairs = 0;
for (Iterator i = v.iterator(); i.hasNext(); ) {
   Integer x = i.next();
   if (x % 2 == 0) somPairs+=x;
}
```

```
Vector<Integer> v = new Vector<Integer>();
v.addElement(13); v.addElement(128); ....
int somPairs = 0;
for (Integer x : v)
   if (x % 2 == 0) somPairs+= x;
```


Objets et Classes

Objet et classe

- *application* = collection d'objets dynamiques en interaction
- *objet* = fournisseur de services utilisés par des *clients*
- programmation par contrat. B. Meyer.
- *classe* = moule à objets
- une classe n'est pas un objet (du moins en Java)
- un objet est une instance d'une classe
- contient attributs (variables et méthodes)

Une première classe

- On souhaite représenter des rectangles en Java

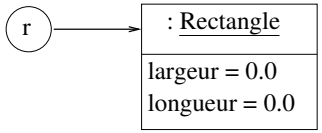
```
public class Rectangle {
  // largeur ⩾ 0, longueur ⩾ 0,
  private double largeur, longueur;
}
```

Créer un objet

```
Rectangle r;
r = new Rectangle();
```

ou

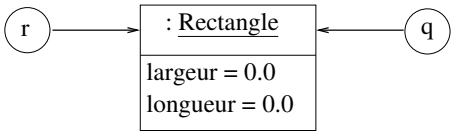
```
Rectangle r = new Rectangle();
```



Références

- En Java, la variable `r` est une référence à l'objet, et non pas l'objet lui-même.
- Pb de l'affectation, de la comparaison et du passage de paramètre.

```
Rectangle q = r;
```



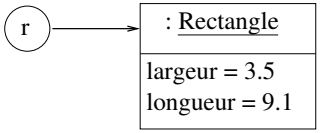
Les constructeurs

- constructeur par défaut (*e.g.* `Rectangle()`)
- valeur initiale `0` (`0.0`, `\u0000`, **`false`**, **`null`**)
- pour initialiser l'objet construit avec des valeurs différentes
- du nom de la classe
- **`this`** désigne l'objet courant et **`this`**`()` son constructeur

```
// construit un Rectangle de largeur l et de longueur L
public Rectangle(double l, double L) {
      this.largeur = l; this.longueur = L;
}
```


Utilisation

```
Rectangle r = new Rectangle(3.5, 9.1);
```



Constructeur par défaut

- Le constructeur par défaut est perdu ! Il faut le redéfinir.

```
// construit un Rectangle de largeur et de longueur égales à 0
public Rectangle() {
      this(0.0, 0.0);
}
```


Destruction des objets

- automatique en Java
- à la charge du support d'exécution
- va dans le sens de la sécurité

Les méthodes

- procédures et fonctions
- objet = automate à états
- une procédure *modifie* l'état
- une fonction *renvoie* une description de l'état
- modèle pas toujours possible

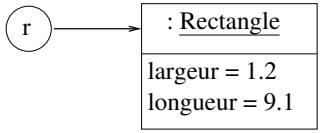
Les méthodes pour Rectangle

```
public class Rectangle {
    private double largeur, longueur;

    // procédure qui modifie la largeur du Rectangle courant
    public void changerLargeur(double l) {
        this.largeur = l;
    }

    // fonction qui renvoie le périmètre du Rectangle courant
    public double périmètre() {
        return 2 * (this.largeur + this.longueur);
    }
}
```

Accès aux attributs et aux méthodes

- notation pointée
- attributs *privés* et *publics*

```
r.largeur;  // erreur car privé
r.périmètre(); // résultat 25.2
r.changerLargeur(1.2);
```



Méthodes et variables statiques

- Existent *indépendamment* de la création des objets
- Accessibles avec le nom de la classe

```
static int partagé;
static void uneMéthode() { ... }
```

`System.out`	*des variables*
`Math.PI`	
`Math.exp(13.4)`	*des méthodes*
`Math.sin(3.14)`	
`main`	

- Les méthodes statiques remettent en cause le modèle objet !

Héritage et liaison dynamique

Héritage

- propriété fondamentale des langages à objets
- outil de la réutilisabilité
- bibliothèques de classes extensibles

Classe héritière (1/4)

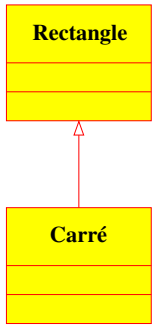
```
public class Carré {
   // Invariant de classe : coté ⩾ 0
   private double coté;

   public Carré(double c) {
       this.coté = c;
   }
   // Rôle : renvoie le périmètre du carré courant
   public double périmètre() {
      return 4*this.coté;
   }
   // Rôle : renvoie la surface du carré courant
   public double surface() {
      return this.coté * this.coté;
   }
} // fin classe Carré
```


Classe héritière (2/4)

- un carré *est un* rectangle particulier
- on souhaite réutiliser la classe Rectangle
- on *hériter* des propriétés de la classe Rectangle

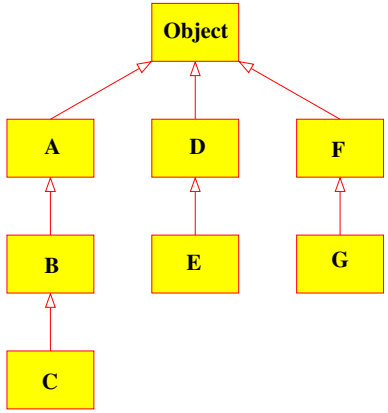
```
public class Carré extends Rectangle {
    // Invariant de classe : largeur == longueur ⩾0
    public Carré(double c) {
        super(c,c);
      }
    // Rôle : met à jour le coté du carré courant
    public void changerCoté(double coté) {
        super.changerLargeur(coté);
        super.changerLongueur(coté);
    }
}
```

Classe héritière (3/4)



- héritage = *spécialisation* et *extension*
- classe mère : **`super`** et **`super()`**
- **`protected`**

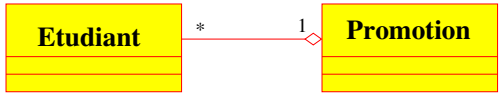
Classe héritière (4/4)



- le graphe d'héritage
- héritage simple ou multiple ?
- la classe `Object`

Héritage ou client

- relation *est-un*
- relation *a-un* (agrégation)



Redéfinition des méthodes (1/2)

- les classes héritières peuvent *redéfinir* des méthodes
- elles doivent posséder la même signature (sinon *surcharge*)
- les classes Rectangle et Carré définissent les méthodes suivantes :

```
// Dans la classe Rectangle
public String toString()
{
  return "rectangle de largeur " + this.largeur
           + " et de longueur " + this.longueur;
}

// Dans la classe Carré
public String toString()
{
  return "carré de coté égal à" + super.largeur;
}
```

Redéfinition des méthodes (2/2)

```
Rectangle r = new Rectangle(2,4);
Carré c = new Carré(6);

System.out.println(r); // r.toString()
System.out.println(c); // c.toString()
```


Recherche d'un attribut ou d'une méthode

- chaque fois que l'on désire accéder à un attribut ou une méthode d'une occurrence d'objet d'une classe C, il devra être défini soit dans la classe C, soit dans l'un de ses ancêtres.
- si l'attribut ou la méthode n'est pas trouvé, c'est une erreur
- s'il y a eu des redéfinitions, sa première apparition en remontant le graphe d'héritage sera choisie.

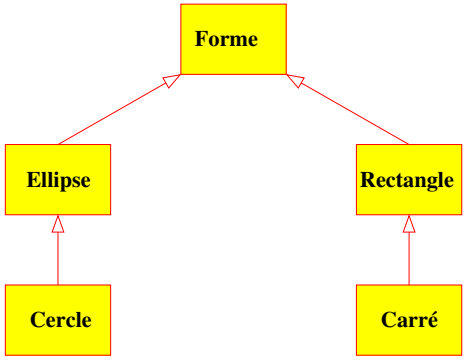
Polymorphisme (1/2)

- langages polymorphiques et non polymorphiques (langages typés et non typés)
- polymorphisme dans les langages de classes est contrôlé par *l'héritage*

```
Rectangle r;
Carré c = new Carré(3);

r = c; // valide : un carré est rectangle
c = r; // erreur de compilation !
       // un rectangle n'est pas (forcément) un carré
```

- Cast : `c = (Carré) r;` si on est sûr que `r` est un Carré
- Exemple une classe `Forme` pour représenter les formes géométriques quelconques

Polymorphisme (2/2)




```
Forme  f;

f = new Rectangle(3,10);
....
f = new Cercle(7);
...
```

Liaison dynamique

```
Forme [] tf = new Formes[10];

tf[0] = new Rectangle(3,10);
tf[1] = new Cercle(7);
...
tf[0] = tf[1]; // tf[0] désigne un cercle ?
...
System.out.println(tf[0]); // toString() de Cercle ?
```

- lorsque qu'il y a des redéfinitions de méthodes, c'est à l'exécution que l'on connaît la méthode à appliquer.
- elle est déterminée à partir de la forme dynamique de l'objet sur lequel elle s'applique.
- Intérêts :
  - localisation
  - évite le pb du switch

Classes et méthodes abstraites

- Une méthode abstraite n'est définie que par sa signature
- Introduite par le mot-clé **abstract**
- Une classe peut être déclarée abstraite, elle décrit des caractéristiques mais pas un fonctionnement
- Une classe abstraite ou qui contient une méthode abstraite ne peut pas être instanciée !

```
public abstract class Forme {
  public abstract int périmètre();
  public abstract int surface();
}
```

Interfaces (1/2)

- pour une forme *simplifiée* de l'héritage multiple
- ce sont des classes dont toutes les méthodes sont *abstraites*
- elles ne possèdent pas d'attribut, à l'exception de constantes
- ne peuvent pas être instanciées

```
public interface MonInterface {
    static final int uneConstate = 10;
    abstract public void uneMéthode(int x);
}
```


Interfaces (2/2)

- une classe peut se *comporter* comme une ou plusieurs interfaces
- Elle doit implanter (**implements**) le ou les interfaces
- elle et ses descendants s'engagent à définir le corps des méthodes de l'interface

```
class MaClasse implements MonInterface {
   public void uneMéthode(int x) {
      ....
   }
}
```

- Serializable
- Cloneable
- Runnable
- ActionListener
- ...

Exception

Le problème

Un événement qui indique une situation anormale pouvant provoquer un dysfonctionnement du programme :

- pb matériel (E/S, mémoire, ...)
- pb logiciel (division par zéro, non respect des invariants, ...)

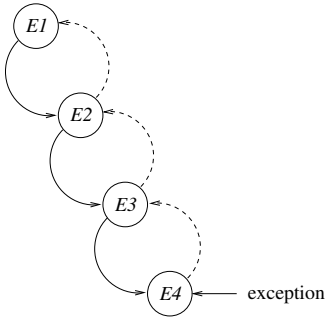
```
tantque B faire
   ...
   si erreur alors
      traiter erreur
   finsi
   ...
fintantque
```

- le code « normal » et le traitement des erreurs mélangés
- code de retour (souvent un entier) *pauvre* en information
- un mécanisme élaboré : *les exceptions*

Traitement

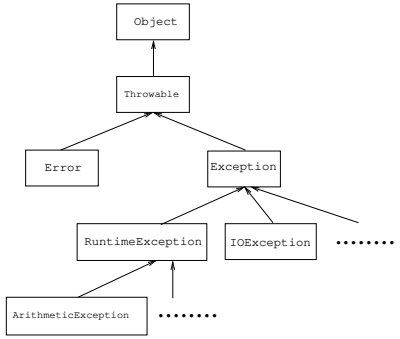
Une exception survient au cours de l'exécution d'une action :

- réexécution de l'action en changeant les conditions initiales d'exécution ;
- transmission de l'exception à l'environnement



Exceptions en Java

- Les exceptions sont des objets
- situation anormale ⇒ émission d'une exception
- java.lang.Exception → java.lang.Throwable



- toutes les exceptions (sauf `RuntimeException`) doivent être explicitement *capturées* ou *déléguées*

Capture d'une exception (1/2)

- clauses **try−catch**

```
try {
    // code « normal »
    ...
    o.m();  // la méthode m peut provoquer une exception
    ...
}
catch (UneException e) {
   // code de traitement de l'exception désignée par e
    ...
}
```

- clause **finally**

```
finally {
   // code exécuté quoiqu'il arrive
}
```


Capture d'une exception (2/2)

```
try {
      int x = StdInput.readlnInt();
      ....
}
catch (IOException e) {
        System.err.println(e);
        System.exit(1);
}
```

```
public static int lirelnInt() {
      try {
            return StdInput.readlnInt();
      }
      catch (IOException e) {
            System.err.println(e);
            System.out.println("réessayer :");
            return lirelnInt();
      }
}
```

Délégation d'une exception

- une méthode n'est pas tenue de capturer une exception
- elle peut déléguer la capture à son environnement
- elle le signale avec la clause **throws** dans son en-tête
- sauf pour les exceptions de type `RuntimeException`

```
public void maMéthode() throws IOException {
   int x = StdInput.readlnInt();
   ....
}
```


Créer ses propres exceptions

```
public class MonException extends Exception {
  int code;
  public MonException() {
      super();
  }
  public MonException(String msg) {
      super(msg);
  }
  public MonException(String msg, int c) {
      super(msg);
      code = c;    // c ≠ 0
  }
  public String toString() {
      return getMessage() + " " + code;
  }
}
```

Émettre une exception

- **throw** (sans s)

```
public void uneMéthode() {
   if (...) {
        throw new ArithmeticException();
   }
   ...
}
```

```
public void maMéthode() throws MonException {
   if (...) {
        throw new MonException("une erreur", 3);
   }
   ...
}
```

- la clause **throws** est obligatoire dans l'en-tête de la méthode qui lève une exception (sauf `RuntimeException`)

Fichiers

Flots d'octets et de caractères

- flots séquentiels unidirectionnels
- flots à accès direct
- octets et caractères Unicode
- quatre classes de base :

	lecture	écriture
flots d'octets	`InputStream`	`OutputStream`
flots de caractères	`Reader`	`Writer`

- paquetage java.io

Fichiers d'octets

Déclaration :

```
FileInputStream is = new FileInputStream("entrée");
FileOutputStream os = new FileOutputStream("sortie");
```

- méthodes de base :
  - `int read()`
  - `void write(int)`
  - `void close()`

Exemple : recopie d'un fichier

```
public void copie (String in, String out)
throws IOException
{
 FileInputStream is = new FileInputStream(in);
 FileOutputStream os = new FileOutputStream(out);

 int c;
 while ((c = is.read()) != -1)
        os.write(c);
  // EOF de is
  is.close();
  os.close();
}
```


Fichier d'objets élémentaires

- `DataInputStream` et `DataOutputstream`
- `readChar`, `writeChar`, `readInt`, `writeInt`, `readDouble`, `writeDouble`, ...
- fin de fichier ⇒ exception `EOFException`

Exemple

```
public double moyenne (String in)
throws IOException {
DataInputStream is = new DataInputStream(
                        new FileInputStream(in));
  int nbÉlem = 0;
  double moy = 0;
  try {
      while (true) {
          moy+= is.readDouble();
          nbÉlem++;
      }
  }
  catch (EOFException e) {
       is.close();
  }
  return nbÉlem == 0 ? 0 : moy / nbÉlem;
}
```


Fichier d'objets

- `ObjectInputStream` et `ObjectOutputstream`
- `Object readObject()`
- **`void`** `writeObject(Object)`
- interface `Serializable`

```
public class Rectangle implements Serializable {
```

Exemple

```
Rectangle r = new Rectangle(3,4);
ObjectOutputStream os = new ObjectOutputStream(
                    new FileOutputStream("Frect"));
// écriture d'un objet de type Rectangle sur le fichier os
os.writeObject(r);
os.close();
ObjectInputStream is = new ObjectInputStream(
                     new FileInputStream("Frect"));
// lecture d'un objet de type Rectangle sur le fichier is
r = (Rectangle) is.readObject();
// ⇐ la conversion nécessaire
is.close();
```


Fichiers de texte

- caractères Unicode
- `FileReader` et `FileWriter`

```
public void copie (String in, String out) throws IOException
{
    Reader is = new FileReader(in);
    Writer os = new FileWriter(out);

    int c;
    while ((c = is.read()) != -1)
        os.write(c);
    // EOF de is
    is.close();
    os.close();
}
```

E/S bufferisées

Pour améliorer les performances, les lectures et les écritures utilisent une zone tampon.

- `BufferedInputStream`, `BufferedOutputStream`
- `BufferedReader`, `BufferedWriter`

```
FileInputStream is =
                  new FileInputStream("entrée");
FileOutputStream os =
                  new FileOutputStream("sortie");

BufferedInputStream bis =
                    new BufferedInputStream(is);
BufferedOutputStream bos =
                    new BufferedOutputStream(os);
```


Passerelles

Les classes `InputStreamReader` et `OutputStreamWriter` sont des passerelles entre les flots d'octets et de caractères.

```
Reader in
   = new BufferedReader(new InputStreamReader(System.in));
//
Writer out
   = new BufferedWriter(new OutputStreamWriter(System.out));
```

Entrée et sortie standard

- `System.in` de type `InputStream`
- `System.out` de type `PrintStream`
- `System.err` de type `PrintStream`

La classe `PrintStream` propose des méthodes `print` et `println` pour écrire tout type d'objet après conversion en chaîne de caractères avec `toString()`
`System.in` ne permet de lire que des octets. Pas de conversion de type implicite ⇒ classe `Scanner` ou classe locale StdInput

```
double x = System.in.read(); // ⇐ suspect
//
Scanner sc = new Scanner(System.in);
double x = sc.nextDouble();
// ou
double x = StdInput.readlnDouble();
```


L'API Java

Application Programming Interface

- ensemble de classes prêtes à l'emploi
- couvre de nombreux aspects de l'informatique
- facilite la construction des applications
- offre des algorithmes efficaces
- organisée en modules et paquetages
- collections
- arithmétique précision infinie
- gestion d'images 2D et3D
- gestion du son
- interfaces graphiques et applet
- processus légers - synchronisation
- réseau (URL, modèle client-serveur RMI, objets distribués IDL Corba)
- beans
- base de données (jdbc)
- réflexivité
- SDK (mécanismes de sécurité)

Collections et Tables

- paquetage `java.util`
- `Vector`, tableaux dynamiques
- `Hashtable`, tables d'adressage dispersé
- `Stack`, piles
- ensembles, listes, arbres
- algorithmes de tri
- algorithmes de recherche
- ...

Collections

- Liste et Ensembles génériques
- Enoncé foreach

```
List<String> vs = new Vector<String>();
SortedSet<String> ts = new TreeSet<String>(vs);
```


La classe `Vector`

tableau dynamique dont la taille est ajustable en fonction des besoins

```
Vector v = new Vector(3);

System.out.println(v.size());        // ⇒ 0
for(int i = 0; i < 10; i++)
    v.addElement(new Integer(i*i));
System.out.println(v.size());        // ⇒ 10
System.out.println(v.elementAt(3)); // ⇒ 9
v.setSize(20);
v.setElementAt(new Integer(3),18);
System.out.println(v.size());        // ⇒ 20
v.removeElementAt(0);
System.out.println(v.size());        // ⇒ 19
v.removeAllElements();
System.out.println(v.size());        // ⇒ 0
```

Tables

- Valeur + Clé

La classe `Hashtable`

Table dont les éléments sont repérés par des *clés* et dont l'accès se fait par adressage dispersé.

```
Hashtable<String,Integer> ht =
                  new Hashtable<String,Integer>();

ht.put("Paul", 3);
ht.put("Pierre",19);
Integer n =  ht.get("Pierre");

if (n != null)
   System.out.println("Pierre = " + n);
```

Les Properties

- sous-classe de `Hashtable`
- correspondance clés et des valeurs `String`
- une instance de `Properties` peut être reliée à une autre instance de `Properties`
- lues et sauvegardées dans un fichier (texte ou XML)

```
Properties p1 = new Properties();
Properties p2 = new Properties(p1);
p1.setProperty("Auteur", "Molière");
p2.setProperty("Compositeur", "Bach");
System.out.print(p1.getProperty("Auteur"));
System.out.print(p2.getProperty("Compositeur"));
System.out.println(p2.getProperty("Acteur","inconnu"));

// propriétés de l'environnement d'exécution
System.out.print(System.getProperty("java.version"));
System.out.print(System.getProperty("os.version"));
System.out.println(System.getProperty("os.arch"));
```


Interfaces Graphiques

- nécessité des interfaces graphiques
- Abstract Window Toolkit (AWT)
- Java Foundation Classes (JFC) – Swing
- applet (little application)
- programmation par événements
- composants graphiques (boutons, menus, canvas, ascenseurs, boîtes de dialogue, etc.)

Énumérations

Énumérations (1/4)

Énumération de constantes représentées par des noms (comme en C), mais c'est bien plus que cela.

```
enum Couleur { trèfle, carreau, coeur, pique }
enum Valeur {deux, trois, quatre, cinq, six, sept,
         huit, neuf, dix, valet, dame, roi, as }

class Carte {
    Couleur couleur;
    Valeur valeur;
    Carte(Valeur v, Couleur c) {
        valeur = v;
        couleur = c;
    }
}

...
new Carte(Valeur.as, Couleur.coeur);

...
```

Énumérations (2/4)

- les énumérations sont des objets
- méthodes prédéfinies `values()`, `ordinal()`

```
for (Couleur c : Couleur.values())
   System.out.println(c);
....
Valeur.quatre.ordinal() // 2
```


Énumérations (3/4)

- on peut ajouter des attributs et des méthodes

```
enum Valeur {
  deux(2), trois(3), quatre(4), cinq(5),
  six(6), sept(7),  huit(8), neuf(9), dix(10),
  valet(10), dame(10), roi(10), as(20);

 int valeur;
 Valeur(int v) { valeur = v; }
 int valeur() { return valeur; }
} // fin enum
```

```
System.out.println(Valeur.as); // as
System.out.println(Valeur.as.valeur()); // 20
```

Énumérations (4/4)

```
// fabrication d'un jeu de 52 cartes
Carte [] jeu52 = new Carte[52];
int i = 0;
for (Couleur c : Couleur.values())
   for (Valeur v : Valeur.values())
       jeu52[i++] = new Carte(v, c);
```


Généricité

Généricité (1/5)

- Permet de paramétrer le type des structures de données
- évite les conversions explicites de type dues à `Object`
- assure les contrôles de type

```
Hashtable<String, Integer> ht = new Hashtable<String, Integer>()
    ;
ht.put("Paul", 3);
ht.put("Pierre", 19);
Integer n = ht.get("Pierre"); // pas de conver. explicite
if (n != null)
    System.out.println("Pierre = " + n);

Vector<Integer> v = new Vector<Integer>();
v.addElement("toto"); // ! erreur compilation
v.addElement(13); v.addElement(128); ...
int somPairs = 0;
for (Integer x : v)
    // pas de conversion explicite
    if (x % 2 == 0) somPairs+= x;
```


Généricité (2/5)

```
class Noeud<T> {
    T val;
    Noeud <T> suivant;
    Noeud(T v) { val = v; }
}

class Pile<E> {
    Noeud <E> sommet;
    void empiler(E x) {
        // ajouter en tête l'élément x
        Noeud<E> p = new Noeud<E>(x);
        p.suivant = sommet;
        sommet = p;
    }
    E sommet() {
        // retourner la valeur du sommet de pile
        return sommet.val;
    }
}
```

Généricité (3/5)

```
// Définition simplifiée d'une Pile générique
class Pile<T> {
   private final int N = 100;
   private int sommet = 0;
   // pas de généricité sur les tableaux !
   private T [] éléments = (T[]) new Object[N];

   public void empiler(T x) {
       éléments[sommet++] = x;
   }

   public void dépiler() {
       sommet--;
   }
   public T sommet() {
       return éléments[sommet-1];
   }
}
```


Généricité (4/5)

```
// une pile de chaînes
Pile<String> p1 = new Pile<String>();
// une pile de piles d'entiers
Pile<Pile<Integer>> p2 =
                  new Pile<Pile<Integer>>();
```

Mais aussi, sans généricité

```
Pile p = new Pile();
p.empiler("bonjour");
p.empiler(6); // OK
// la conversion suivante est obligatoire
Integer i = (Integer) p.sommet();
```

mais alors, message du compilateur

```
Note: Test.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Généricité (5/5)

Généricité sur plusieurs types :

```
class Noeud<C,V> {
 C clé;
 V val;
 Noeud <C,V> suivant;
 Noeud (C c, V v) { clé = c; val = v; }
}
```

et avec les interfaces :

```
interface Pile<T> { void empiler(T x);
}

// Implémentation d'une pile par un tableau
class PileTableau<E> implements Pile<E> { ... }
// idem avec une structure chaînée
class PileChaînée<E> implements Pile<E> { ... }
...
Pile<String> p = new PileChaînée<String>();
```


Généricité et Polymorphisme (1/2)

```
Pile<Integer> p1 = new Pile<Integer>();
Pile<Object> p2 = p1;
```

Illégal : car on pourrait empiler autre chose que des Integer !
De même :

```
void afficherSommet(Pile<Object> p) {
    System.out.println(p.sommet());
}
...
Pile<Integer> p = new Pile<Integer>();
afficherSommet(p); // types incompatibles
```

Généricité et Polymorphisme (2/2)

Solution : la notation `?`

```
void afficherSommet(Pile<?> p) {...}
...
Pile<Integer> pi = new Pile<Integer>();
afficherSommet(pi); // OK
Pile<String> ps = new Pile<String>();
afficherSommet(ps); // OK
```

- pour limiter le polymorphisme aux sous-classes de `A`

```
void afficherSommet(Pile<? extends A> p) {...}
```

- `? super A` pour limiter le polymorphisme aux super-classes de `A`

Méthodes génériques

```
void dupliquerSommet(Pile<?> p) {
   Object o = p.sommet(); // OK
   p.empiler(o); // erreur de compilation
   // car le type des éléments est inconnu
}
```

Solution : méthodes génériques

```
<T> void dupliquerSommet(Pile<T> p) {
   T o = p.sommet();
   p.empiler(o);
}
```

```
Pile<Integer> p1 = new Pile<Integer>();
Pile<String> p2 = new Pile<String>();
dupliquerSommet(p1);
dupliquerSommet(p2);
```

Fonction anonymes

Introduction

- depuis la version 8 de Java
- API fortement récrite + nouveautés (Stream)
- modèle fonctionnel (lisp, haskell, ...)
- fonction sans nom (anonyme)

$$(x, y) \rightarrow x + y$$

- valeur fonctionnelle :

```
(x,y) -> x + y
(int x, int y) -> x + y
```

Type d'une fonction anonyme (1/3)

C'est une interface *fonctionnelle* (**une seule** méthode abstraite)

```
interface FoncInt {
    int apply(int x, int y);
}
```

```
FoncInt f = (x,y) -> x + y;
FoncInt f = (x,y) -> x * x + y;
```


Type d'une fonction anonyme (2/3)

```
(x) -> { assert x>=0;
         return Math.sqrt(x);
       }

() -> {
        System.out.print("n = ");
        int n = StdInput.readlnInt();
        System.out.println(n*n);
      }
```

```
interface FoncDouble {
  double apply(double x);
}

interface Procédure {
    void apply() throws IOException;
}
```

Type d'une fonction anonyme (3/3)

Les fonctions anonymes peuvent être génériques

```
interface BiFunction<T, U, R> {
  R apply(T t, U u);
}

BiFunction<Integer,Integer,Integer> f = (x,y) -> x + y;
```

- `java.util.function`

Utilisation

- Évaluation (inférence de types)

```
FoncInt f = (x,y) -> x + y;
System.out.println(f.apply(2,3)); // 5
```

- Lambda en paramètre

```
public double calculerAire(double a,
     double b,int n,Function<Double, Double> f)
```

- Lambda résultat de fonction

```
// (g∘f)(x) = g(f(x))
<T,R,Z> Function<T,Z> compose(Function<T,R> f,
                              Function<R,Z> g) {
    return x -> g.apply(f.apply(x));
}
```

Fermeture

- Occurrences liées *vs* libres
- En Java, la portée est *statique* (environement de définition de la fct anonyme)

```
interface Procédure { void apply(); }
...
int x=1;
Procédure p = () -> { System.out.println(x); };
p.apply();
```

- Les variables libres sont implicitement **final** (*i.e.* elles ne peuvent donc pas être modifiées)

```
Procédure p = () -> { System.out.println(x++); };
// ERREUR DE COMPILATION ! ! ! ! ! ! ! ! ! !
```


Méthodes par défaut

- plusieurs méthodes dans une interface fonctionnelle ?
- pour garantir la compatibilité avec l'existant (*i.e.* Iterable, Runnable, ...)
- méthodes par défaut (**default**)

```
public interface Iterable<T> {

  // méthode à implémenter
   public Iterator<T> iterator();

   // méthode par défaut
   public default void forEach(Consumer<? super T> action) {
       for (T t : this)
            action.accept(t);
   }
}
```

Threads

Introduction

- processus = programme en cours d'exécution
- exécution (quasi) parallèle des processus
- processus léger (thread)
- la classe `Thread` (`java.lang`)
- la méthode `run`
- un thread pour `main`
- application terminée lorsque *tous* les threads sont finis
- priorité (`getPriority()` et `setPriority(p)`)

La classe Thread

```
class MonThread extends Thread {
    // constructeur
    public MonThread() { ... }
    // code à exécuter par le thread
    public void run() { ... }
}
/**
 * Classe de Test qui crée et exécute 2 threads
 */
public class TestThread {
    public static void main(String [] arg) {
        MonThread threadA = new MonThread(),
                  threadB = new MonThread();
        threadA.start() // et non pas run()
        threadB.start();
    }
}
```


L'interface Runnable (1/2)

- pas d'héritage multiple ⇒
  interface *fonctionnelle* `Runnable`
- `Thread t1 = new Thread(t2)`

```
class MonThread implements Runnable {
    public MonThread() { ... }
    public void run() { ... }
}
....
MonThread tA = new MonThread();
MonThread tB = new MonThread();
new Thread(tA).start();
new Thread(tB).start();
```

```
Runnable r = () -> System.out.println("1");
new Thread(r).start();
// ci-dessous le type Runnable est déterminé par inférence
new Thread(() -> System.out.println("2")).start();
```

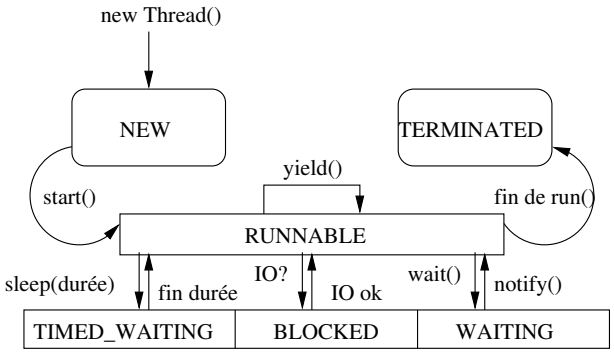
L'interface Runnable (2/2)

```
public class Aléatoire implements Runnable {
    public Thread proc = new Thread(this);
    private Random rand;
    private long intervalle;
    public Aléatoire(long germe, long time) {
        rand = new Random(germe);
        intervalle = time * 1000;
    }
    public void run() {
        while (true) {
            try { Thread.sleep(intervalle); }
            catch (InterruptedException e) {}
            System.out.print(rand.nextInt() + " ");
            System.out.flush();
        }
    }
} // fin classe Aléatoire
```


L'interface Runnable (2/2)

```
public class TestProd {
    public static void main(String [] args) {
        new Aléatoire(10, 1).proc.start();
        new Aléatoire(15, 3).proc.start();
        new Aléatoire(13, 3).proc.start();
    }
}
```

États d'un threads (1/2)



États d'un threads (2/2)

`BLOCKED`	*thread en attente de la libération d'une donnée.*
`NEW`	*thread qui n'a pas encore commencé à s'exécuter.*
`RUNNABLE`	*thread exécutable, prêt à disposer de la puissance de calcul.*
`TERMINATED`	*thread dont l'exécution est terminée.*
`TIMED_WAITING`	*thread en attente avec une durée maximum spécifiée.*
`WAITING`	*thread en attente.*

Interruption d'un processus (1/2)

- méthode *interrupt()*
- le processus peut vérifier s'il est interrompu ou non (il gère l'interruption)

```
public void run() {
    while (!Thread.interrupted()) {
        try {
            Thread.sleep(intervalle);
        }
        catch (InterruptedException e) {
            // interruption lors du sleep
            break;
        }
        System.out.print(rand.nextInt() + " ");
        System.out.flush();
    }
}
```


Interruption d'un processus (2/2)

```
public class TestProd {
   public static void main(String [] args) {
      new Aléatoire(10, 1).proc.start();
      new Aléatoire(15, 3).proc.start();
      Aléatoire p = new Aléatoire(13, 1);
      p.proc.start();
      ...
      // arrêt du 3ème thread
      p.proc.interrupt();
  }
}
```

- interruption dans le sleep ⇒ exception
- hors du sleep ⇒ drapeau Thread.interrupted()
- getPriority() et setPriority(int)

Attente de la fin d'un thread

- méthodes `join()` ou `join(long ms)` (pour les timeout)

```
public static void main(String [] args) {
    Aléatoire p1 = new Aléatoire(10, 1);
    Aléatoire p2 = new Aléatoire(15, 2);
    Aléatoire p3 = new Aléatoire(13, 1);

    p1.proc.start();
    p2.proc.start();
    p3.proc.start();
    try {
        p1.proc.join(); // attendre la fin de p1
        p2.proc.join(); // attendre la fin de p2
        p3.proc.join(); // attendre la fin de p3
    }
    catch (InterruptedException e) {}
    ...
}
```


Gestion des threads (1/4)

static Thread	currentThread() *Renvoie une référence sur le thread actif.*
static void	dumpStack() *Affiche une trace de la pile d'exécution du thread courant.*
String	getName() *Renvoie le nom du thread.*
int	getPriority() *Renvoie la priorité du thread.*
Thread.State	getState() *Renvoie l'état courant du thread.*

Gestion des threads (2/4)

static boolean	`interrupted()` *Renvoie* **true** *si le thread est interrompu.*
boolean	`isAlive()` *Teste si le thread est actif.*
void	`setName(String name)` *Change le nom du thread.*
void	`setPriority(int newPriority)` *Change la priorité du thread.*
void	`interrupt()` *Interrompt le thread.*

Gestion des threads (3/4)

void	`join()` ou `join(long millis)` ou `join(long millis, int nanos)` *Attend que le thread meurt, au plus le temps indiqué en millisecondes et nanosecondes si ces paramètres sont fournis.*
void	`notify()` *Réveille le thread précédemment mis en attente.*
void	`run()` *Cette méthode par défaut ne fait rien. Si le thread est issu d'un objet de type* `Runnable`*, c'est la méthode* `run` *de cet objet qui est exécutée ou sinon sa redéfinition éventuelle.*

Gestion des threads (4/4)

static void	`sleep(long millis)` ou `sleep(long millis, int nanos)` *Provoque l'endormissement du thread pour une durée égale au nombre de millisecondes auquel s'ajoutent les éventuelles nanosecondes.*
void	`start()` *Provoque le démarrage du thread, la JVM appelle alors sa méthode* `run` *.*
void	`wait()` ou `wait(long timeout)` ou `wait(long timeout, int nanos)` *Met le thread en attente d'un* `notify` *invoqué par un autre processus ou du dépassement du temps passé en paramètre.*
void	`yield()` *Force le scheduling, le thread cède sa place aux autres processus éligibles.*

Synchronisation des threads (1/3)

- pb de l'accès à une ressource critique (*i.e.* compte bancaire)
- pose d'un verrou
- toutes les méthodes qui *modifient* l'état d'un d'objet partagé **doivent** être déclarées **`synchronized`**.

```
class CompteBancaire {
  private double solde;
  public synchronized
                    void opération(double somme)
  {
     if (solde+somme<0)
        System.err.println("retrait impossible");
     else
        solde+=somme;
  }
}
```

Synchronisation des threads (2/3)

- **synchronized** (obj) énoncé ;

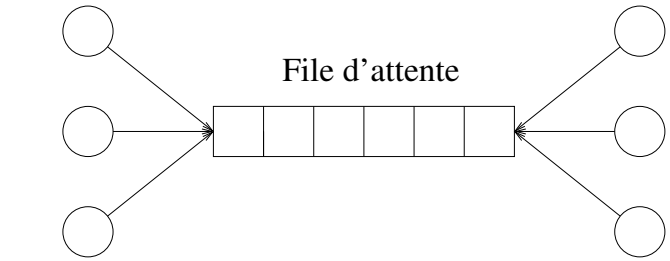
ainsi une méthode qualifiée `synchronised` peut se récrire :

```
...
void uneMéthodeÀSynchroniser() {
  synchronized (this) {
     ... // code du corps de la méthode
  }
}
...
```


Synchronisation des threads (3/3)

```
...
synchronized (...) { // début du bloc contrôlé
  ... // actions diverses
  // tests de vérification des données
  wait(); // les données ne sont pas prêtes
  // attente d'un notify() transmis par ailleurs

  // réveil et reprise du cours de l'exécution
  ... // actions sur les données

  // le besoin de verrouiller n'est plus
  notify() // message de réveil des processus endormis
  ... // actions diverses
} // fin du bloc
...
```

Producteurs Consommateurs (1/2)



- producteurs et consommateurs doivent se synchroniser (file vide)

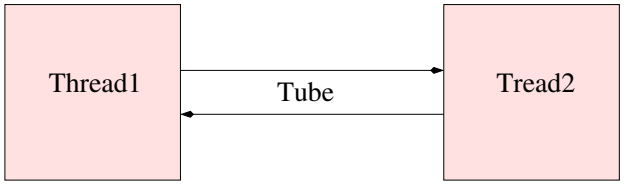
Producteurs Consommateurs (2/2)

```
import java.util.*;
public class File<T> extends Vector<T> {
    public synchronized void enfiler(T e) {
        addElement(e);
        notify();
    }
    public synchronized T défiler() {
        if (isEmpty())
            try {
                wait();
            }
            catch (InterruptedException e) {
                return null;
            }
        T e = elementAt(0);
        removeElementAt(0);
        return e;
    }
} // fin classe File
```

Tubes

Tubes (1/2)

- Un *tube* est un canal bidirectionnel entre 2 threads
- Les threads peuvent *écrire* et *lire* dans le tube



- La synchronisation Producteur/Consommateur est gérée par le support d'exécution

Tubes (2/2)

- `PipedInputStream` et `PipedOutputStream`
- les tubes d'entrée et de sortie doivent être connectés :
  - à la création, avec le constructeur

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = null;
try {
     po = new PipedOutputStream(pi);
}
catch (IOException e) { System.err.println(e); }
```

- avec la méthode `connect()`

```
PipedInputStream pi = new PipedInputStream();
PipedOutputStream po = new PipedOutputStream();
try {
     pi.connect(po);
}
catch (IOException e) { System.err.println(e); }
```


Réseau

Connexion à une URL

- paquetage `java.net`
- classe `java.net.url`
- URL (Uniform Ressource Locator)

  protocole`://[utilisateur@]`*hôte*`[:`*port*`][/`*chemin/fichier*`]`

```
URL(adr)
URL(prot,host,file)
getProtocol()
getHost()
getFile()
URLConnection openConnection()
setDoInput() , setDoOutput()
connect()
openStream()  // pour la lecture
getContent()
```


Connexion à une URL (2/3)

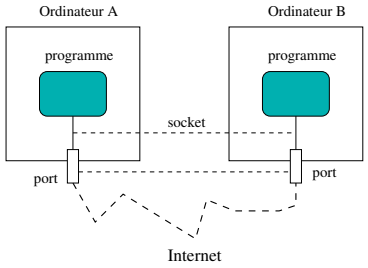
```
URL u = new URL("ftp://vg:secret@taloa.unice.fr/in/f");
URLConnection uc = u.openConnection();
// indiquer l'accès en écriture
uc.setDoOutput(true);
// créer un flot de sortie
OutputStream os = uc.getOutputStream();
FileInputStream is = new FileInputStream("f");

// copie le fichier f.txt
int b;
while ((b = is.read()) != -1)
    os.write((byte) b);

// fin de fichier
is.close(); os.close();
```

Connexion à une URL (3/3)

```
public class AfficherUrl {
  public static void main(String[] args) throws IOException {
   try {
       URL url = new URL(args[0]);
       System.out.println(url.getHost()+ "/" + url.getFile());
       // convertir l'url en InputStream
       BufferedReader in = new BufferedReader(new
           InputStreamReader(url.openStream()));
       String inputLine;
       while ((inputLine = in.readLine()) != null)
            System.out.println(inputLine);
       in.close();
    }
    catch (MalformedURLException|UnknownHostException e) {
         System.err.println("Bad Url: " + args[0]);
    }
  }
} // fin classe AfficherUrl
```


Socket

- services (http, ftp, telnet, ...) sont associés à des *ports*
- un port est une *entrée virtuelle* de l'ordinateur
- pour obtenir un service, il faut *(host, port)*
- dans un programme, on établit une *ligne virtuelle* entre les ports des ordinateurs à connecter grâce à une *socket*.



- port = entier sur 16 bits. Ports jusqu'à 1024 réservés (http 80, telnet 23, ...), au dessus, libres
- un programme peut gérer plusieurs ports et plusieurs sockets

Modes de transmission

Communication par commutateurs :

1. Mode *non connecté* :
   - commutation par paquets indépendants,
   - chaque paquet possède les adresses de départ et d'arrivée
   - Internet UDP, *datagrams*
   - *multicast*
   - connexion non fiable, ordre non garanti
2. Mode *connecté* :
   - circuit virtuel de commutation établi au préalable
   - Internet TCP
   - connexion fiable, ordre garanti

Datagrammes (émission 1/2)

- classe `DatagramPacket` définit les paquets à transmettre
- classe `InetAddress` définit les adresses Internet

```
InetAddress adrSource = InetAddress.getLocalHost();
InetAddress adrDest = InetAddress.getByName("djinn");
```

Un paquet Datagram contient l'information à transmettre :

```
String msg = "blabla blabla";
byte[] data = msg.getBytes();
DatagramPacket paquet = new DatagramPacket(data, data.length,
    adrDest, port);
```

Pour envoyer le paquet, il faut une socket :

```
DatagramSocket socket = new DatagramSocket();
socket.send(paquet);
```

Datagrammes (émission 2/2)

```
public class Emetteur {
  public static void main (String [] args) throws IOException {
  Scanner sc = new Scanner(System.in);
  // adresse du destinataire
  InetAddress adrDest = InetAddress.getByName(args[0]);
  int portDest = Integer.parseInt(args[1]);
  // création de la socket
  DatagramSocket socket = new DatagramSocket();
  byte [] data;
  DatagramPacket paquet = null;
  while (sc.hasNext()) {
   System.out.print("> ");
   String msg = sc.nextLine();
   // convertir le message en tableau de bytes
   data = msg.getBytes();
   // créer le paquet et l'expédier
   paquet=new DatagramPacket(data,data.length,adrDest,portDest);
   socket.send(paquet);
  }
  socket.close();}}
```


Datagrammes (réception 1/2)

Un paquet Datagram contenant l'information à recevoir doit être créé :

```
byte [] data = new byte [512];
DatagramPacket paquet = new DatagramPacket(data,data.length);
```

Pour recevoir le paquet, il faut également une socket avec le port :

```
DatagramSocket socket = new DatagramSocket(port);
socket.receive(paquet);
```

Datagrammes (réception 2/2)

```
public class Receveur {
 public static void main (String [] args) throws IOException {
  int port = Integer.parseInt(args[0]);
  // création de la socket
  DatagramSocket socket = new DatagramSocket(port);
  // tampon où mettre le message reçu
  byte [] data = new byte [512];
  // créer un paquet de réception des messages
  DatagramPacket paquet=new DatagramPacket(data,data.length);
  while (true) {
   // recevoir le prochain message
   socket.receive(paquet);
   // afficher le message reçu
   System.out.print("from "+paquet.getAddress().getHostName());
   String msg=new String(paquet.getData(),0,paquet.getLength());
   System.out.println(" " + msg);
  }
 }
}
```


Datagrammes (exécution)

Sur l'ordinateur émetteur, lyre :

```
lyre$ javac Emetteur.java
lyre$ java  Emetteur djinn 17000
> blabla blabla
> abracadabra
>
```

Sur l'ordinateur récepteur, djinn :

```
djinn$ javac  Receveur.java
djinn$ java  Receveur 17000
from lyre.polytech.unice.fr blabla blabla
from lyre.polytech.unice.fr abracadabra
```

Datagrammes - Multicast

- Pour envoyer un message à un groupe de machines destinatrices
- `MulticastSocket`
- les paquets sont envoyés à une adresse multicast : 224.0.0.1 – 239.255.255.255 à laquelle doit se joindre tous les destinataires

```
MulticastSocket mcs = new MulticastSocket(port);
InetAddress mcsAdr = InetAddress.getByName("228.5.6.7");
mcs.joinGroup(mcsAdr);
```

- Les datagrammes sont ensuite normalement expédiés ou reçus avec les méthodes `send` et `receive` précédentes
- quitter le groupe : `mcs.leaveGroup(mcsAdr);`
- exemple d'utilisation : programme `chat`

Mode connecté TCP (émission 1/2)

Un émetteur se connecte sur la machine destinatrice grâce à une socket :

```
Socket socket = new Socket("djinn", portDest);
// ou bien
Socket socket = new Socket();
// 1er port libre
socket.bind(new InetSocketAddress("localhost",0));
socket.connect(new InetSocketAddress("djinn", portDest));
```

Un flot est créé entre les deux machines.

```
DataOutputStream out=new DataOutputStream(socket.getOutputStream
    ());
DataInputStream in=new DataInputStream(socket.getInputStream());
```

L'émetteur écrit (ou lit) dans le fichier `out` (ou `in`).

Mode connecté TCP (émission 2/2)

```
public class Emetteur {
 public static void main (String [] args) throws IOException {
   Scanner sc = new Scanner(System.in);
   int portDest = Integer.parseInt(args[1]);
   // création de la socket sur le 1er port libre
   Socket socket = new Socket();
   socket.bind(new InetSocketAddress("localhost", 0));
   // Connexion avec la machine distante
   socket.connect(new InetSocketAddress(args[0], portDest));
   System.out.println(socket.getLocalAddress());
   PrintStream out = new PrintStream(socket.getOutputStream());
   while (sc.hasNext()) {
     System.out.print("> ");
     out.println(sc.nextLine());
   }
   out.close();
   socket.close();
 }
} // classe Emetteur
```


Mode connecté TCP (réception 1/2)

- Le receveur doit créer une socket spéciale, `ServerSocket` en spécifiant le numéro de port
- Il indique ensuite qu'il est prêt à accepter la connexion d'un émetteur

```
ServerSocket socketServeur = new ServerSocket(port);
Socket socketEmetteur = socketServeur.accept();
```

Mode connecté TCP (réception 2/2)

```
public class Receveur {
  public static void main (String [] args) throws IOException {
    int port = Integer.parseInt(args[0]);
    // création du socket serveur
    ServerSocket socketServeur = new ServerSocket(port);
    // création du socket default réception
    Socket socket = socketServeur.accept();
    // création du flot de communication
    Scanner sc = new Scanner(socket.getInputStream());
    while (sc.hasNext()) {
      System.out.print("from " + socket.getInetAddress().
          getHostName()+" ");
        System.out.println(sc.nextLine());
    }
  }
} // fin classe Receveur
```


Mode connecté TCP (exécution)

Sur l'ordinateur émetteur, lyre :

```
lyre$ javac Emetteur.java
lyre$ java  Emetteur djinn 17000
> blabla blabla
> abracadabra
>
```

Sur l'ordinateur serveur, djinn :

```
djinn$ javac  Receveur.java
djinn$ java  Receveur 17000
Serveur actif sur le port 17000
from lyre.polytech.unice.fr blabla blabla
from lyre.polytech.unice.fr abracadabra
```

Modèle Client/Serveur (1/4)

Dans un modèle *client/serveur*, le receveur est le serveur qui peut accepter *plusieurs* connexions de différents clients. Chaque client est traité par dans un thread.

```
while (true) {
  // attendre la prochaine connexion
  Socket socketClient = socketServeur.accept();
  new GestionDuClient(socketClient); // thread
}
```


Modèle Client/Serveur (2/4)

```
public class Serveur {
 public static void main (String [] args) throws IOException {
   int port = Integer.parseInt(args[0]);
   // creation de la socket
   ServerSocket socketServeur = new ServerSocket(port);
   System.out.println("Serveur actif " + "sur le port " + port);
   while (true) {
     // attendre la prochaine connexion
     Socket socketClient = socketServeur.accept();
     System.out.println(socketClient.getInetAddress().
         getHostName() + " connecté");
     // gérer le client dans un thread
     new GestionDuClient(socketClient);
   }
 }
}
```

Modèle Client/Serveur (3/4)

```
class GestionDuClient implements Runnable {
  public Thread proc = new Thread(this);
  private Socket socket;
  private BufferedReader din;

  public GestionDuClient(Socket s) throws IOException {
    socket = s;
    InputStreamReader in = new InputStreamReader(s.
        getInputStream());
    din = new BufferedReader(in);
    proc.start();
  }

  public void run() {
     .....
  }
}
```


Modèle Client/Serveur (4/4)

```
class GestionDuClient implements Runnable {
  ....

 public void run() {
  while (true)
   try {
     System.out.print("from " +  socket.getInetAddress().
         getHostName());
     // on lit ligne par ligne
     System.out.println(" " + din.readLine());
   }
   catch (IOException e) { break; }
   System.out.println(socket.getInetAddress().getHostName() + "
       déconnecté");
  // fermer la socket
  try { socket.close();}
  catch (IOException e) {}
 }
}
```

Modèle Client/Serveur (exécution 1/2)

Sur les ordinateurs clients

```
lyre$ javac Emetteur.java

lyre$ java  Emetteur djinn 17000
> blabla blabla
> abracadabra
>

mondrian$ javac Emetteur.java

mondrian$ java  Emetteur djinn 17000
> hello ...
>
```


Modèle Client/Serveur (exécution 2/2)

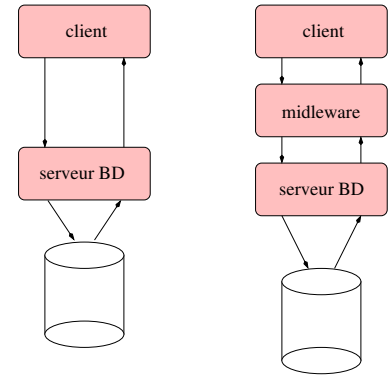
Sur l'ordinateur serveur, djinn :

```
djinn$ javac  Serveur.java

djinn$ java  Serveur 17000
Serveur actif sur le port 17000
mondrian.polytech.unice.fr connecté
lyre.polytech.unice.fr connecté
from lyre.polytech.unice.fr blabla blabla
from mondrian.polytech.unice.fr hello ...
from lyre.polytech.unice.fr abracadabra
```

JDBC

Introduction

- Java Database Connectivity
- offre la connexion d'applications Java avec différents types de BD (SQL, tableurs, etc.)
- API JDBC est indépendante du SGBD.
- Ensemble de classes (`java.sql.*`)



Modele 2 couches Modele 3 couches

Pilote

L'accès à la BD se fait par l'intermédiaire d'un pilote.

1. un pont entre JDBC et ODBC (Microsoft). La communication se fait *via* des fonctions C appelées par les méthodes du pilote ;
2. les pilotes recourant à des fonctions non-Java spécifiques du SGBD ;
3. les pilotes permettent l'emploi d'un serveur middleware. Il s'agit d'une interface avec le SGBD *via* une API spécifique ;
4. les pilotes Java propriétaires utilisant directement le protocole réseau du SGBD (*e.g.* mysql-connector).

Structuration

- `import java.sql.*`
- JDBC gère :
  - la connexion à la BD
  - l'envoi de requêtes SQL
  - l'exploitation des résultats provenant de la BD

Connexion

`DriverManager` :

- gère les pilotes chargés *(ici pour mysql)* :

```
try { // créer une nouvelle instance du Driver
   // Class.forName("com.mysql.jdbc.Driver").newInstance() ;
   Class.forName("com.mysql.cj.jdbc.Driver").
       getDeclaredConstructor().newInstance();
} catch (Exception ex) { /* gérer les erreurs */ }
```

- crée les connexions (`Connection`) :

```
String urlBD="jdbc:mysql://localhost/bd";
try (Connection conn =
    DriverManager.getConnection(urlBD,"user","pass");
)
{
  // code de l'application qui accède à la BD
  ...
} catch (Exception ex) { /* gérer les erreurs */ }
```


Statement

- Envoi de requêtes SQL (`Statement`)
- `createStatement` pour une requête SQL simple (sans paramètre)
- `prepareStatement` pour une requête SQL (précompilée) avec paramètres
- `prepareCall` pour l'appel d'une procédure préparée.
- `ResultSet executeQuery()`, `int executeUpdate()`

```
// créer un Statement
Statement stmt = conn.createStatement();
// exécuter une requête SQL
ResultSet rs = stmt.executeQuery("SELECT * FROM uneTable;");
```

ResultSet

- `ResultSet` résultat d'une requête SQL (*i.e.* lignes de la table)
- méthodes `get` pour accéder aux colonnes de la ligne courante
- `next()` pour passer le *curseur* à la ligne suivante
- `previous()`
- conversion de types entre SQL et Java (`getString`, `getInt`, `getDate`, ...)

```
rs.absolute(2);
System.out.println("Resultat : " + rs.getInt(1));

rs.beforeFirst();
while (rs.next()) {
  String nom = rs.getString("nom");
  String espèce = rs.getString("espece");
  Date naissance = rs.getDate("naissance");
  System.out.println(nom + " " + espèce + " " + naissance);
}
```


prepareStatement

- requête précompilée (plus efficace)
- paramètres

```
// INSERT INTO animal VALUES
// -> ('Coco','Diane','Perroquet','f','2000-06-09',NULL) ;

PreparedStatement pstmt =
   conn.prepareStatement("INSERT INTO animal VALUES (?,?,?,?,?,
       NULL);");

// définir les paramètres
pstmt.setString(1, "Coco");
pstmt.setString(2, "Diane");
// exécuter la requête
pstmt.executeUpdate();
```

callStatement

- requête précompilée pour appeler une fonction de la BD
- paramètres

```
CallableStatement cstmt = conn.prepareCall("call myfunc(?,?);");
// fixer la valeur du 1er paramètre donnée (e.g. un réel)
cstmt.setFloat(1, 34.5f);
// le 2ème paramètre est un résultat (e.g. une chaîne)
cstmt.registerOutParameter(2,Types.VARCHAR);
// exécuter la requête
cstmt.execute();
// afficher le le résultat
System.out.println(cstm.getString(2));
```