# 3

## Les classes fondamentales

1. Présentation des API
2. Les enveloppes des types de base
3. La classe `java.lang.Object` mère de toutes les classes
4. Les chaînes de caractères
5. Outils mathématiques
6. Ensembles structurés, itérateurs et comparateurs
7. Introspection





## Les API

Les API (Application Programming Interface) forment l'interface de programmation, c'est-à-dire l'ensemble des classes livrées avec Java.

| | |
|---|---|
| `java.lang` | classes de base du langage |
| `java.io` | entrées / sorties |
| `java.util` | ensemble d'outils : les classes très "util" |
| `java.net` | classes réseaux |
| `java.applet` | classes pour les appliquettes |
| `java.awt` | interfaces graphiques (Abstract Windowing Toolkit) |

et de nombreuses autres.



## Les enveloppes des types de base

| | |
|---|---|
| `boolean` | java.lang.Boolean |
| `char` | java.lang.Character |
| `byte` | java.lang.Byte |
| `short` | java.lang.Short |
| `int` | java.lang.Integer |
| `long` | java.lang.Long |
| `float` | java.lang.Float |
| `double` | java.lang.Double |

- Une instance de la classe enveloppe encapsule une valeur du type de base correspondant.
- Chaque classe enveloppe possède des méthodes pour extraire la valeur d'un objet. Par exemple `intValue()` renvoie un `int`. Les noms varient d'une classe à l'autre.
- Un objet enveloppant est immutable : la valeur contenue ne peut être modifiée.
- On transforme souvent une valeur en objet pour utiliser une méthode manipulant ces objets.



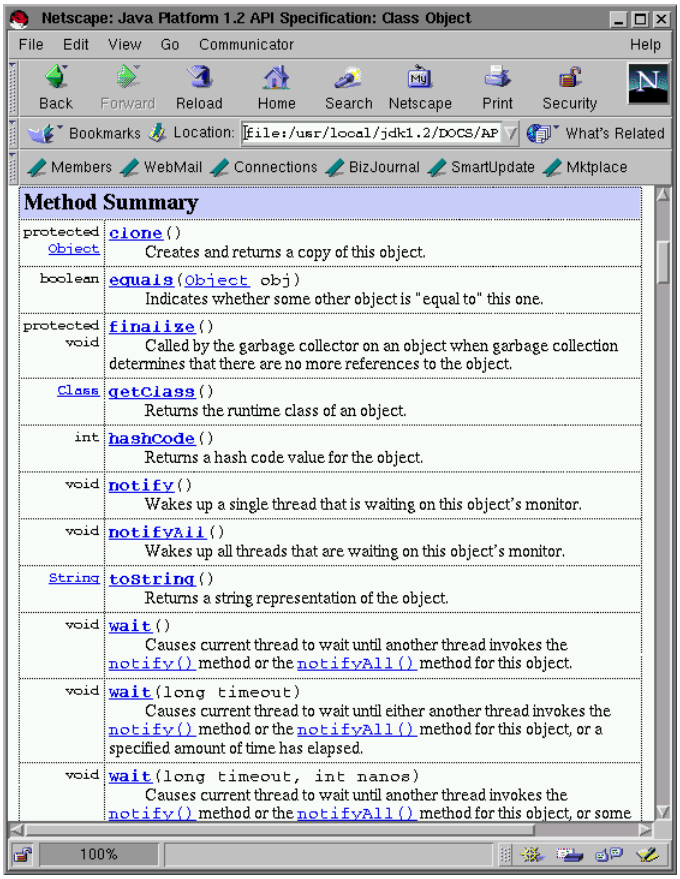
## La classe `java.lang.Object`

### Affichage d'un objet et hashcode

La méthode `toString()` retourne la représentation d'un objet sous forme de chaîne de caractères (par défaut le nom de la classe suivi de son *hashcode*) :

```
System.out.println(new Integer(3).toString());
//affiche 3
```

La valeur du hashcode peut être obtenue par la méthode `hashcode()` de la classe `Object`.






## Le clonage

Le clonage est la construction d'une copie d'un objet.
La classe `Object` contient une méthode `Object clone()`.

- Quand elle est appelée sur un objet d'une classe qui implémente l'interface `Cloneable`, elle crée une copie de l'objet du même type. La copie est superficielle de l'objet : les attributs de l'objet sont alors recopiés.
- Quand elle est appelée sur un objet d'une classe qui n'implémente pas l'interface `Cloneable`, elle lève l'exception `CloneNotSupportedException`.
- La class `Object` n'implémente pas l'interface `Cloneable`!

Exemple:

```
public class Point implements Cloneable {
  private int x, y;

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

  public String toString() {
    return(this.x + ", " + this.y);
  }

  public static void main(String[] args)
    throws CloneNotSupportedException
  {
    Point a = new Point(3,5);
    Point b = (Point) a.clone(); // méthode clone() de Object
    b.x = a.x + 1;
```



```
    System.out.println(a); // 3 5
    System.out.println(b); // 4 5
  }
}
```

Si on veut une classe clonable et une classe dérivée non clonable, la classe dérivée implémente `Cloneable` mais on lève une exception dans l'écriture de `clone()`.

### Exemple de clonage

Le clonage par défaut est superficiel.

```
public class Pile implements Cloneable {
   int hauteur;
   int[] contenu;

   public Pile (int maxP) {
     hauteur = 0;
     contenu = new int[maxP];
   }

  public void push(int val) {
    contenu[hauteur++]=val;
  }

  public int pop() {
    return contenu[--hauteur];
  }

  public Object clone() throws CloneNotSupportedException {
    Pile nouvelObjet = (Pile) super.clone();
    nouvelObjet.contenu = (int[]) contenu.clone();
    return nouvelObjet;
  }
}
```



```
public class TestPile{
  public static void main(String[] args) {
    Pile f = new Pile(2);
    f.push(5);
    f.push(6);
    try {
      Pile g = (Pile) f.clone();
      System.out.println(g.pop());  // 6
      System.out.println(g.pop());  // 5
    } catch (CloneNotSupportedException e) {}
  }
}
```

Remarquer l'utilisation de `super.clone()` qui appelle `clone()` de `Object` crée toujours un objet du bon type. L'appel à `clone()` sur un objet d'une classe dérivée de `Pile` serait incorrect si on avait utilisé `new Pile()`.
Exemple

```
public class DPile extends Pile {
  int cout;

  public DPile (int maxP) {
    super(maxP);
    cout = 0;
  }

  public Object clone() throws CloneNotSupportedException  {
    DPile nouvelObjet = (DPile) super.clone();
    nouvelObjet.cout = cout;
    return nouvelObjet;
  }
}
```

## Égalité entre objets

La méthode `equals()` de la classe `Object` détermine si deux objets sont équivalents. Par défaut deux objets sont équivalents s'ils sont accessibles par la même référence.
Une classe peut redéfinir la méthode `equals()`.

```
class Rectangle extends Forme{
  int largeur, hauteur;
  Rectangle(int largeur, int hauteur) {
    this.largeur = largeur;
    this.hauteur = hauteur;
  }
  public boolean equals(Object arg) {
    if (!(arg instanceof Rectangle))
      return false;
    Rectangle rarg = (Rectangle)arg;
    return (largeur == rarg.largeur)
      && (hauteur == rarg.hauteur);
  }
```

Remarquer que l'argument est de type `Object`. Si l'argument était `Rectangle`, la méthode serait surchargée. Elle serait alors ignorée lors d'un appel avec un argument de type `Forme` qui référence un `Rectangle`. La comparaison entre les deux rectangles serait alors incorrecte.



## Les chaînes de caractères

*La classe* `java.lang.String` :

- La classe `String` est `final` (ne peut être dérivée).
- Elle utilise un tableau de caractères (membre privé de la classe).
- Un objet de la classe `String` ne peut être modifié. (On doit créer un nouvel objet).

```
String nom = "toto" + "tata";
System.out.println(nom.length()); // 8
System.out.println(nom.charAt(2)); // t
```

On peut construire un objet `String` à partir d'un tableau de caractères :

```
char tableau = {'t','o','t','o'};
String s = new String(tableau);
```

et inversement :

```
char[] tableau = "toto".toCharArray();
```

Conversion d'un entier en chaîne de caractère :

```
String un = String.valueOf(1); // methode statique
                               qui appelle toString()
```

et inversement :

```
int i = Integer.valueOf("12").intValue(); // ou bien :
int i = Integer.parseInt("12");
```



Comparaison des chaînes de caractères : on peut utiliser la méthode `equals()` :

```
String s = "toto";
String t = "toto";
if (s.equals(t)) ...  // true
```

La méthode `compareTo()` est l'équivalent du `strcmp()` du C.

La méthode `trim()` supprime tous les espaces blancs :

```
String s = "   toto     ";
s = s.trim();
```

*La classe* `java.lang.StringBuffer`

- permet de créér des buffers de caractères de taille extensible.

```
StringBuffer sb = new StringBuffer("le");
sb.append(" petit ");
sb.append("prince");
```

On récupère un `String` à partir d'un `StringBuffer` (sans recopie) :

```
String mot = sb.toString();
```

Ces classes sont sécurisées au niveau des *thread* (voir plus tard).



*La classe* `java.util.StringTokenizer` permet d'effectuer de l'analyse lexicale simple.

```
String texte = "le petit prince";
StringTokenizer st = new StringTokenizer(texte);
while (st.hasMoreTokens())
   { String mot = st.nextTokens();}
```

La classe `StringTokenizer` implémente l'interface `java.util.Enumeration`.
Les délimiteurs par défaut sont les espaces, retours chariot et tabulations. On peut spécifier les délimiteurs :

```
String texte = "http://www-igm.univ-mlv.fr/~beal";
StringTokenizer st = new StringTokenizer(texte,"/:~");
if (st.countTokens() < 2) ...  // mauvaise URL
String protocole = st.nextToken();
String host = st.nextToken();
String login = st.nextToken();
```

## Outils mathématiques

On peut trouver des outils mathématiques dans les deux classes et le paquetage suivants :

- `java.lang.Math`
- `java.util.Random`
- `java.math` (pour le travail sur des entiers ou flottants longs)

Exemple : `int maximum = Math.max(3,4);`

Exemple : Tirer au hasard un entier entre 100 et 1000 (les deux compris).

```
int maximum = 100 + (int)(Math.random()*901);
```

- Une instruction arithmétique sur les entiers peut lever l'exception `ArithmeticException` :

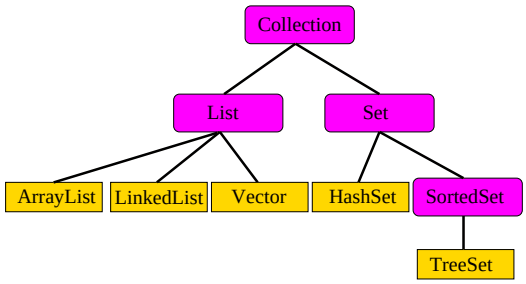
```
try { int i = 1/0;}
catch (ArithmeticException e) {...}
```

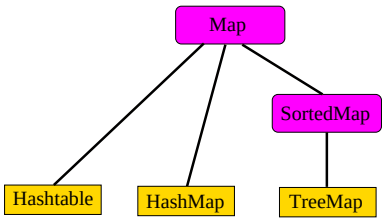
- Une instruction arithmétique sur les flottants (`double`) ne lève pas d'exception. Une expression flottante peut prendre trois valeurs particulières :

| | |
|---|---|
| `POSITIVE_INFINITY` | $1.0/0.0$ |
| `NEGATIVE_INFINITY` | $-1.0/0.0$ |
| `NaN` | $0.0/0.0$ `// Not a Number` |



## Ensembles structurés, itérateurs et comparateurs






Regardons par exemple :

- `java.util.ArrayList`

```
$ javap java.util.ArrayList
public class java.util.ArrayList
extends java.util.AbstractList implements java.util.List,
java.lang.Cloneable, java.io.Serializable {
    public java.util.ArrayList();
    public java.util.ArrayList(int);
    public void add(int, java.lang.Object);
    public boolean add(java.lang.Object);
    public void clear();
    public java.lang.Object clone();
    public boolean contains(java.lang.Object);
    public java.lang.Object get(int);
    public int indexOf(java.lang.Object);
    public boolean isEmpty();
    public java.lang.Object remove(int);
    public java.lang.Object set(int, java.lang.Object);
    public int size();
}
```

- `java.util.Iterator`

```
$ javap java.util.Iterator
public interface java.util.Iterator {
    public abstract boolean hasNext();
    public abstract java.lang.Object next();
    public abstract void remove();
}
```



## Vue d'ensemble

Deux interfaces

- `Collection` pour les ensembles d'objets,
- `Map` pour les tables, c'est-à-dire des ensembles de couples (clé, valeur), où la clé et la valeur sont des objets.

Des itérateurs sur les collections

- `Iterator` interface des itérateurs,
- `ListIterator` itérateur sur les séquences.
- `Enumeration` ancienne forme des itérateurs.

De plus, deux classes d'utilitaires

- `Collections` avec algorithmes de tri etc,
- `Arrays` algorithmes spécialisés pour les tableaux.

Les opérations principales sur une collection

- `add` pour ajouter un objet,
- `remove` pour enlever un élément,
- `contains` test d'appartenance,
- `size` pour obtenir le nombre d'éléments,
- `isEmpty` pour tester si l'ensemble est vide.

Comme les éléments sont des objets, ne pas écrire `c.add(2)` mais `c.add(new Integer(2))`.

Sous-interfaces spécialisées de `Collection`

- `List` spécifie les *séquences*, avec les méthodes
  - `int indexOf(Object o)` position de `o`
  - `Object get(int i)` retourne l'objet à la position i.
  - `Object set(int i, Object o)` remplace l'élément en position i, et retourne l'élément qui y était précédemment.
- `Set` spécifie les ensembles sans duplication.
- `SortedSet` sous-interface de `Set` pour les ensembles ordonnés.
  - `Object first()` retourne le premier objet.
  - `Object last()` retourne le dernier objet.
  - `SortedSet subset(Object initial, Object final)` retourne une référence vers le sous-ensemble des objets $\geq$ `initial` et $<$ `final`.

Opérations ensemblistes sur les collections

- `boolean containsAll(Collection c)` pour tester l'inclusion.
- `boolean addAll(Collection c)` pour la réunion.
- `boolean removeAll(Collection c)` pour la différence.
- `boolean retainAll(Collection c)` pour l'intersection.

Les trois dernières méthodes retournent `true` si elles ont modifié la collection.



## Implémentation d'une collection

Pour les collections

- `ArrayList` (recommandée, par tableau), et `LinkedList` (par liste doublement chaînée) implémentent `List`.
- `Vector` est une vieille classe (JDK 1.0) "relookée" qui implémente aussi `List`. Elle a des méthodes personnelles.
- `HashSet` (recommandée) implémente `Set`.
- `TreeSet` implémente `SortedSet`.

Le choix de l'implémentation résulte de l'efficacité recherchée : par exemple, l'accès indicé est en temps constant pour les `ArrayList`, l'insertion entre deux éléments est en temps constant pour les `LinkedList`.

Discipline d'abstraction:

- les attributs, paramètres, variables locales sont déclarés avec, comme type, une interface (`List`, `Set`)
- les classes d'implémentation ne sont utilisées que par leur constructeurs.

Exemple

```
List l = new Arraylist();
Set s = new HashSet();
```



Exemple : Programme qui détecte une répétition dans les chaînes de caractères d'une ligne.

```
import java.util.Set;
import java.util.HashSet;

class TestSet {
  public static void main(String[] args) {
    Set s = new HashSet();
    for (int i = 0; i < args.length; i++)
      if (!s.add(args[i]))
        System.out.println("Déjà vu : " + args[i]);
    System.out.println(s.size() + " distincts : " + s);
  }
}
```

```
$ java TestSet a b c a b d
Déjà vu : a
Déjà vu : b
4 distincts : [d, c, b, a]
```



## Itérateurs

L'interface `Iterator` définit les itérateurs.

Un *itérateur* permet de parcourir l'ensemble des éléments d'une collection.

Java 2 propose deux schémas, l'interface `Enumeration` et l'interface `Iterator`.

L'interface `java.util.Iterator` a trois méthodes

- `boolean hasNext()` qui teste si le parcours contient encore des éléments;
- `Object next()` qui retourne l'élément suivant, si un tel élément existe (et lève une exception sinon).
- `void remove()` qui supprime le dernier élément retourné par `next`.

L'interface `java.util.Enumeration` a deux méthodes

- `boolean hasMoreElements()` qui teste si l'énumération contient encore des éléments;
- `Object nextElements()` qui retourne l'élément suivant, si un tel élément existe (et une exception sinon).

```
import java.util.*;

class Personne {
  String nom;
  int age;
  public Personne(String nom, int age) {
    this.nom = nom; this.age = age;
  }
  public String toString() { return "Nom: "+nom+", age: "+age; }
```

```
}

class TestHashSet {
  public static void printAll(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
      System.out.println(i.next());
  }
  public static void main(String[] args) {
    Set s = new HashSet();
    s.add(new Personne("Pierre", 23));
    s.add(new Personne("Anne", 20));
    s.add("Université");
    s.add("Marne-la-Vallée");
    printAll(s);
    Set t = (Set) ((HashSet) s).clone();
    System.out.println(s.size());
    printAll(t);
    Iterator i = t.iterator();
    while(i.hasNext())
      if (i.next() instanceof Personne) i.remove();
    printAll(t);
  }
}
```

Avec les résultats

```
$ java TestHashSet
Marne-la-Vallée
Nom: Pierre, age: 23
Nom: Anne, age: 20
Université
4
Marne-la-Vallée
Nom: Pierre, age: 23
Nom: Anne, age: 20
Université
Marne-la-Vallée
Université
```

Observer le désordre.



Détails sur les itérateurs.

- la méthode `iterator()` de la collection positionne l'itérateur au "début",
- la méthode `hasNext()` teste si l'on peut progresser,
- la méthode `next()` avance d'un pas dans la collection, et retourne l'élément *traversé*.
- la méthode `void remove()` supprime l'élément référencé par `next()`, donc pas de `remove()` sans `next()`.

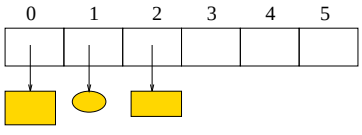
```
|A B C     iterator(), hasNext() = true
 A|B C     next() = A, hasNext() = true
 A B|C     next() = B, hasNext() = true
 A B C|    next() = C, hasNext() = false

Iterator i = c.iterator();
i.remove(); // NON
i.next();
i.next();
i.remove(); // OK
i.remove(); // NON
```



### Exemple de tableau

On désire créér un tableau des références sur des objets de type `Forme` qui peuvent être `Rectangle` ou `Ellipse` (voir chapitre 1).

```
import java.util.ArrayList;
import java.util.Iterator;

abstract class Forme {
  abstract double getAire(); // a définir
  String toStringAire() {
    return "aire = " + getAire();
  } // commune
}
class Rectangle extends Forme{ ... }
class Ellipse extends Forme { ... }

class TestForme {
  public static void main(String[] args) {
    Forme r1 = new Rectangle(6,10);
    Forme r2 = new Rectangle(5,10);
    Forme e = new Ellipse(3,5);
    List liste = new ArrayList();
    liste.add(r1);
    liste.add(r2);
    liste.add(1,e);  // on a r1, e, r2
    for (Iterator it = liste.iterator(); it.hasNext();)
      System.out.println( ((Forme) it.next()).toStringAire());
  }
}
```






### Itérer sur les listes

Les listes sont des séquences. Un ***iterateur de listes*** implémente l'interface `ListIterator`. Il a des méthodes supplémentaires:

- `Object previous()` qui permet de reculer, joint à
- `boolean hasPrevious()` qui retourne vrai s'il y a un élément qui précède.
- `void add(Object)` qui ajoute l'élément juste avant l'itérateur.
- `void set(Object o)` qui substitue `o` à l'objet référencé par `next()`

```
import java.util.*;

class TestLinkedList {
  public static void printAll(Collection c) { ... }
  public static void main(String[] args) {
    List a = new LinkedList();
    a.add("A");
    a.add("B");
    a.add("C");
    printAll(a);  // A B C
    ListIterator i = a.listIterator();
    System.out.println(i.next()); // A | B C  -> A
    System.out.println(i.next()); // A B | C  -> B
    System.out.println(i.hasPrevious()); // true
    System.out.println(i.previous());  // A | B C -> B
    i.add("X");
    printAll(a);  // A X | B C
  }
}
```

## Comparaison

Java exprime que les objets d'une classe sont comparables, en demandant que la classe implémente l'interface `Comparable`.

L'interface `Comparable` déclare une méthode `int compareTo(Object o)` telle que `a.compareTo(b)` est

- négatif, si `a < b`.
- nul, si `a = b`.
- positif, si `a > b`.

Exemple : comparaisons de "noms".

```
import java.util.*;

class Nom implements Comparable {
  private String nom;
  private int age;

  public Nom(String nom, int age) {
    this.nom = nom; this.age = age;
  }
  public int compareTo(Object o) {
    Nom n = (Nom) o;
    int comp = nom.compareTo(n.nom);
    return (comp !=0) ? comp : n.age - age;
  }
  public String toString() { return nom + " : " + age; }
}
```



```
class TestComparaison {
  public static void main(String[] args) {
    Collection c = new TreeSet();
    c.add(new Nom("Paul", 21));
    c.add(new Nom("Paul", 25));
    c.add(new Nom("Anne", 25));
    for (Iterator i = c.iterator(); i.hasNext(); )
      System.out.println(i.next());
  }
}
```

avec le résultat

```
$ java TestComparaison
Anne : 25
Paul : 25
Paul : 21
```



## Comparateur

Un *comparateur* est un objet qui permet la comparaison. En Java, l'interface `Comparator` déclare une méthode `int compare(Object a, Object b)`

On se sert d'un comparateur

- dans un constructeur d'un ensemble ordonné.
- dans les algorithmes de tri fournis par `Collections`.

Exemple de deux comparateurs de noms:



```
class NomComparator implements Comparator {
  public int compare(Object oa, Object ob) {
    Nom a = (Nom) oa;
    Nom b = (Nom) ob;
    int comp = a.nom().compareTo(b.nom());
    if (comp == 0)
      comp = a.age() -b.age();
    return comp;
  }
}
class AgeComparator implements Comparator {
  public int compare(Object oa, Object ob) {
    Nom a = (Nom) oa;
    Nom b = (Nom) ob;
    int comp = b.age() - a.age();
    if (comp == 0)
      comp = a.nom().compareTo(b.nom());
    return comp;
  }
}
```

Une liste de noms (pour pouvoir trier sans peine).

```
public static void main(String[] args) {
  List c = new ArrayList();
  c.add(new Nom("Paul", 21));
  c.add(new Nom("Paul", 25));
  c.add(new Nom("Anne", 25));
  printAll(c);
  Collections.sort(c, new NomComparator());
  printAll(c);
  Collections.sort(c, new AgeComparator());
  printAll(c);
}
```

Et les résultats:

```
Paul : 21  Paul : 25  Anne : 25  // ordre d'insertion
Anne : 25  Paul : 21  Paul : 25  // ordre sur noms
Anne : 25  Paul : 25  Paul : 21  // ordre sur ages
```

## Tables

L'interface `Map` spécifie les tables, des ensembles de couples (clé, valeur). Les clés ne peuvent être dupliquées, au plus une valeur est associée à une clé.

- `Object put(Object key, Object value)` insère l'association `(key, Object value)` dans la table et retourne l'objet précédemment associé à la clé ou bien `null`
- `boolean containsKey(Object key)` retourne vrai s'il y a un objet associé à cette clé.
- `Object get(Object key)` retourne l'objet associé à la clé dans la table.
- `Object remove(Object key)` supprime l'association de clé `key`. Retourne l'objet précédemment associé. Retourne `null` si `null` était associé ou si `key` n'est pas une clé de la table.

La sous-interface `SortedMap` spécifie les tables dont l'ensemble des ***clés*** est ordonné.

Implémentation d'une table

Pour les tables

- `HashMap` (recommandée), implémente `Map`.
- `Hashtable` est une vieille classe (JDK 1.0) "relookée" qui implémente aussi `Map`. Elle a des méthodes personnelles.
- `TreeMap` implémente `SortedMap`.

La classe `TreeMap` implémente les opérations avec des arbres rouge-noir.

Un `TreeMap` stocke les références de ces objets de telle sorte que les opérations suivantes s'exécutent en temps $O(\log(n))$ :



- `boolean containsKey(Object key)`
- `Object get(Object key)`
- `Object put(Object key, Object value)`
- `Object remove(Object key)`

pourvu que l'on définisse un bon ordre. C'est `java.util.Comparator` permet de spécifier un comparateur des clés.





### Formes nommées

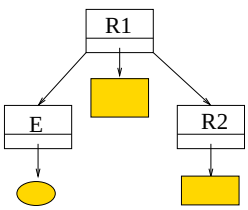
On associe un nom à chaque forme. Le nom est la *clé* de la forme.

L'interface `Forme`, et les classes `Rectangle` et `Ellipse` sont comme d'habitude.

```
import java.util.*;

public class FormeMapTest {
  public static void main(String[] args) {
    Forme r1 = new Rectangle(6,10);
    Forme r2 = new Rectangle(5,10);
    Forme e = new Ellipse(3,5);
    TreeMap arbre  = new TreeMap();
    arbre.put("R2",r2);
    arbre.put("R1",r1);
    arbre.put("E",e);
    System.out.println( ((Forme)arbre.get("R1")).toStringAire());
  }
}
```

On obtient :

```
$ java TestForme
aire = 60.0
```






Si l'on désire trier les clés en ordre inverse, on change le comparateur.

La classe `java.util.TreeMap` possède un constructeur qui permet de changer le comparateur :

```
public java.util.TreeMap(java.util.Comparator);
```

Le programme devient :

```
import java.util.*;
public class OppositeComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    return - ((Comparable) o1).compareTo(o2);  //ordre inverse
  }
}
```

Cette méthode lève une `NullPointerException` si `o1` est `null`. Le reste de la vérification est délégué à `compareTo`.

```
class TestForme {
  public static void main(String[] args) {
    Forme r1 = new Rectangle(6,10);
    Forme r2 = new Rectangle(5,10);
    Forme e = new Ellipse(3,5);
    Comparator c = new OppositeComparator();
    TreeMap arbre  = new TreeMap(c);
    arbre.put("R2",r2);
    arbre.put("R1",r1);
    arbre.put("E",e);
    System.out.println(arbre.firstKey() + " " + arbre.lastKey());
    // "R2" "E"
  }
}
```

## Itérer dans les tables

Les tables n'ont pas d'itérateurs.
Trois méthodes permettent de *voir* une table comme un ensemble

- `keySet()` retourne l'ensemble (`Set`) des clés;
- `values()` retourne la collection des valeurs associées aux clés;
- `entrySet()` retourne l'ensemble des couples (clé, valeur);

```
Map m = ...;
Set clés = m.keySet();
Set couples = m.entrySet();
Collection valeurs = m.values();
```

On peut ensuite itérer sur ces ensembles:

```
for (Iterator i = clés.iterator(); i.hasNext(); )
  System.out.println(i.next());

for (Iterator i = valeurs.iterator(); i.hasNext(); )
  System.out.println(i.next());

for (Iterator i = couples.iterator(); i.hasNext(); ) {
  Map.Entry e = (Map.Entry) i.next();
  System.out.println(e.getKey() + " -> " + e.getValue());
```



## Exemple : construction d'un index

On part d'une suite d'entrées formées d'un mot et d'un numéro de page, comme

```
22, "Java"
23, "Itérateur"
25, "Java"
25, "Map"
25, "Java"
29, "Java"
```

et on veut obtenir un "index", comme

```
Itérateur [23]
Java [22, 25, 29]
Map [25]
```

Chaque mot apparaît une fois, dans l'ordre alphabétique, et la liste des numéros correspondants et donnée en ordre croisssant, sans répétition.



```
import java.util.*;

class Index extends TreeMap {
  public void put(int page, String mot) {
    Set numeros = (Set) get(mot);
    if (numeros == null) {
      numeros = new TreeSet();
      put(mot, numeros); // la ''vraie'' méthode put
    }
    numeros.add(new Integer(page));
  }
  public void print() {
    Set clés = keySet();
    for (Iterator i = clés.iterator(); i.hasNext(); ) {
      String c = (String)i.next();
      System.out.println(c + " " + get(c));
    }
  }
}

class TestIndex {
  public static Index makeIndex() {
    Index index = new Index();
    index.put(22,"Java");
    index.put(23,"Itérateur");
    index.put(25,"Java");
    index.put(25,"Map");
    index.put(25,"Java");
    index.put(29,"Java");
    return index;
  }
  public static void main(String[] args) {
    Index index = makeIndex();
    index.print();
  }
}
```



## Algorithmes

Les classes `Collections` et `Arrays` (attention au "s" final) fournissent des algorithmes dont la performance et le comportement est garanti. Toutes les méthodes sont statiques.

`Collections`:

- `min`, `max`, dans une collection d'éléments comparables;
- `sort` pour trier des listes (tri par fusion);

```
List a;
...
Collections.sort(a);
```

- `binarySearch` recherche dichotomique dans les listes ordonnées.
- `copy` copie de listes, par exemple,

```
List source = ...;
List dest;
Collections.copy(dest, source);
```

- `synchronizedCollection` pour "synchroniser" une collection : elle ne peut être modifiée durant l'exécution d'une méthode.

`Arrays`:

- `binarySearch` pour la recherche dichotomique, dans les tableaux;
- `equals` pour tester l'égalité des contenus de deux tableaux, par exemple

```
int[] a, b ...;
boolean Arrays.equals(a,b);
```

- `sort` pour trier un tableau (quicksort), par exemple

```
int[] a;
...
Arrays.sort(a);
```

Exemple : tirage de loto.

```
import java.util.*;

class Loto {
  public static void main(String[] args) {
    List boules = new ArrayList(49);
    for (int i=0; i < 49; i++)
      boules.add(new Integer(i));
    Collections.shuffle(boules); // mélange
    List tirage = boules.subList(0,6); // les 6 premières
    Collections.sort(tirage); // tri
    System.out.println(tirage); // et les voici
  }
}
```

Résultat:

```
> java Loto
[6, 17, 24, 33, 41, 42]
> java Loto
[15, 24, 28, 41, 42, 44]
> java Loto
[27, 30, 35, 42, 44, 46]
```



## Introspection

La classe `java.lang.Class` :

- permet de manipuler les classes et interfaces comme des objets;
- offre des possibilités d'introspection (exploration des méthodes et constructeurs d'une classe).

On peut ensuite récupérer un objet "méthode". Les classes de ces objets sont définies dans `java.lang.reflect` :

```
String s = "toto";
Class c = s.getClass();
interface I {}
Class c1 = I.class;
Class c2 = Class.forName("I");
Class c3 = float.class;
```

Une instance de la classe `Class` est associée à toutes les classes, interfaces, tableaux ou types primitifs.

On peut appliquer les méthodes suivantes à un objet `c` de la classe `Class`

- `getDeclaredMethods()` retourne un tableau d'objets de la classe `java.lang.reflect.Method`, les méthodes déclarées dans `c`;
- `getMethods()` retourne aussi les méthodes héritées;
- `getMethods(String, Class[] parameterTypes)` recherche une méthode en fonction de son profil.

Une méthode de la classe `Method` peut ensuite être invoquée par `invoke()`.



```
import java.lang.reflect.*;
import java.util.Date;

public class Test{
  public static void main(String[] args) throws Exception {
    Class classeDate = Class.forName("java.util.Date"); // ou "Date"
    Object maDate = classeDate.newInstance();
      //une instance de la classe Date est créée
    Method maSortie = classeDate.getMethod("toString", null);
      //retourne la methode toString() de la classe Date
    System.out.println((String) maSortie.invoke(maDate, null));
      //appel de toString() sur maDate
    Date aujourdhui = new Date();
    System.out.println(aujourdhui);
    System.out.println(maDate);
  }
}
```

On obtient :

```
> java Test
Thu Sep 09 14:54:09 CEST 1999
Thu Sep 09 14:54:09 CEST 1999
Thu Sep 09 14:54:09 CEST 1999
```



## Le chargement de classes

Un *chargeur de classes* (*classloader*) est une instance d'une sous-classe de la classe abstraite `java.lang.ClassLoader`. Il charge le bytecode d'une classe à partir d'un fichier `.class` et la rend accessible aux autres classes.

Principe du fonctionnement de la méthode `loadClass()` de la classe `ClassLoader` :

1. appel à `findLoadedClass()` pour voir si la classe n'est pas dejà chargée;
2. demande de chargement de la classe à un chargeur parent obtenu par `getParent()`;
3. en cas d'échec, appel de la méthode `findClass()`;
4. levée de l'exception `ClassNotFoundException` en cas de nouvel échec.

```
public Class loadClass(String name)
       throws ClassNotFoundException {
  try {
    Class c = findLoadedClass(name);
    if (c != null) return c;

    ClassLoader parent = getParent();
    try {
      c = parent.loadClass(name);
      if (c != null) return c;
    } catch (ClassNotFoundException e) {}

    c = findClass(name);
    if (c !=null) return c;
  } catch (Exception e) {
      throw new ClassNotFoundException(name);
    }
}
```

La méthode `findClass()` appelle une méthode `defineClass()` qui est la méthode de base de tout chargeur de classes. Elle

- crée une instance de la classe `Class`
- stocke la classe dans le chargeur

La signature de `defineClass()` est :

```
Class defineClass(String name,byte[] b,int off,int len)
                throws ClassFormatError
```



```
import java.io.*;

public class VerboseClassLoader extends ClassLoader{
  public VerboseClassLoader(){
    super(getSystemClassLoader());//chargeur parent en param.
  }
  public Class loadClass(String name)
    throws ClassNotFoundException {
    System.out.println("Chargement de "+ name);
    try {
      byte[] b = loadClassData(new File(name+".class"));
      return defineClass(name,b,0,b.length);
    } catch (Exception e) {
        return getParent().loadClass(name);
      }
  }
  private byte[] loadClassData(File f) throws IOException {
    FileInputStream entree = new FileInputStream(f);
    int length = (int)f.length();
    int offset = 0;
    int nb;
    byte[] tableau = new byte[length];
    while (length != 0) {
      nb = entree.read(tableau,offset,length);
      length -= nb;
      offset += nb;
    }
    return tableau;
  }
```



```
  public static void main(String[] args) throws Exception{
    VerboseClassLoader cl = new VerboseClassLoader();
    Class clazz = cl.loadClass("A");
    Object o = clazz.newInstance();
    System.out.print("Dans VerboseClassLoader : ");
    if (o instanceof A)
      System.out.println("o instance de A");
    else
      System.out.println("o n'est pas instance de A");
    System.out.println((o.getClass()).getClassLoader());
    A o2 = new A();
    System.out.println((o2.getClass()).getClassLoader());
  }
}
```

Étant données trois classes vides `B`,`C` et `D`, la classe `A` est :

```
public class A extends B {
  C c;
  D d;

  public A() {
    System.out.println("nouveau A()");
    d = new D();
  }

  public void inutile(){
    c = new C();
  }
}
```



Dans l'exemple précédent,

- l'appel à `newInstance()` crée un objet et charge les classes nécessaires à sa création;
- `o` et `o2` *n'appartiennent pas* à la même classe : deux classes de même nom (ici `A`) peuvent coexister dans la machine virtuelle si elles n'ont pas le même chargeur de classe. Ceci est important pour la *programmation réseau*.

On obtient :

```
> java VerboseClassLoader
Chargement de A
Chargement de B
Chargement de java.lang.Object
Chargement de java.lang.Throwable
Chargement de java.lang.System
Chargement de java.io.PrintStream
nouveau A()
Chargement de D
Dans VerboseClassLoader : o n'est pas instance de A
VerboseClassLoader@4acca8a
nouveau A()
sun.misc.Launcher$AppClassLoader@1714ca8a
```

# 4

## La programmation concurrente





## Programmation concurrente

La *programmation concurrente* est l'ensemble des mécanismes permettant l'exécution concurrente d'actions spécifiées de façon séquentielle.
En Java, deux mécanismes permettent un ordonnancement automatique des traitements :

- la concurrence entre commandes du système (processus);
- la concurrence entre processus légers de la machine virtuelle.



## Processus légers

Un *processus léger* (*thread*) correspond à un fil d'exécution (une suite d'instruction en cours d'exécution). Il s'agit d'un processus créé et géré par la machine virtuelle Java.

S'il y a plusieurs processus légers :

- ils sont associés à un même programme;
- ils s'exécutent dans le même espace mémoire.

Lorsque l'on parle de processus léger, il y a trois notions bien distinctes :

- un objet représentant le code à exécuter (la cible).
  La classe de cet objet implémente l'interface `Runnable`.
- un objet qui contrôle du processus léger.
  Il est d'une classe dérivant de `Thread`.
- un fil d'exécution, c'est-à-dire la séquence d'instructions en cours d'exécution.
  C'est le code de la méthode `run()` de la cible.

Ne pas confondre `Thread` (le contrôleur) et `Runnable` (le contrôlé). C'est d'autant plus facile que `Thread` implémente `Runnable` et peut donc s'autocontrôler !



## La classe `java.lang.Thread`

- Un objet de la classe `Thread` ne représente pas un processus léger mais un *objet de contrôle du processus léger*.
- Au lancement d'un programme, la machine virtuelle possède un unique processus léger qui exécute le `main()` de la classe appelée.

```
public class MaThread{
  public static void main(String[] args) throws Exception{
    Thread threadInitiale = Thread.currentThread();
    threadInitiale.setName("Thread initiale");
    System.out.println(threadInitiale);
    Thread.sleep(1000);
    System.out.println(threadInitiale.isAlive());
    Thread maThread = new Thread();
    maThread.setName("Ma thread");
    System.out.println(maThread);
    System.out.println(maThread.isAlive());
  }
}
```

On obtient à l'exécution :

```
Thread[Thread initiale,5,main]
true
Thread[Ma thread,5,main]
false
```

Chaque processus léger appartient à un groupe de processus légers (ici `main`) et a une priorité (ici 5).

## Démarrage et terminaison

- **Démarrage** d'un processus léger par la méthode `start()` du thread.
- **Exécution** du processus léger par le thread qui appelle la méthode `run()` du runnable.

La méthode `run()` peut être spécifiée de deux façons différentes :

- en implémentant la méthode `run()` de l'interface `Runnable` (solution explicite).
- en redéfinissant la méthode `run()` de la classe `Thread` (solution directe).

Le processus léger se termine à la fin du `run()`.

La classe `Thread` possède sept constructeurs qui spécifient :

- le nom du processus léger (par défaut `Thread-i`);
- le groupe du processus léger (un objet de la classe `ThreadGroup`);
- la cible (target) du processus léger : un objet implémentant l'interface `Runnable` qui précise la méthode `run()` à exécuter lors du démarrage du processus léger.



## Le lapin et la tortue (première version)

Classe des lapins

```
public class Lapin implements Runnable{
  public void run() {
    long t = System.currentTimeMillis(), x = t;
    for (int i = 0; i<5; i++){
      x = System.currentTimeMillis();
      System.out.println("Lapin "+i
        + " au temps "+ (x-t) + " ms.");
      try {
        Thread.sleep(300);  // il se repose peu
      } catch(InterruptedException e) {}
    }
    x = System.currentTimeMillis();
    System.out.println("Lapin est arrivé au temps "
      + (x-t) + " ms.");
  }
}
```

Classe des tortues

```
public class Tortue implements Runnable{
  public void run() {
    long t = System.currentTimeMillis(), x = t;
    for (int i = 0; i<5 ; i++){
      x = System.currentTimeMillis();
      System.out.println("Tortue "+i
        + " au temps "+ (x-t) + " ms.");
      try {
        Thread.sleep(500);   // il se repose beaucoup
      } catch(InterruptedException e) {}
    }
    x = System.currentTimeMillis();
    System.out.println("Tortue est arrivée au temps "
      + (x-t) + " ms.");
  }
}
```



Mise en place

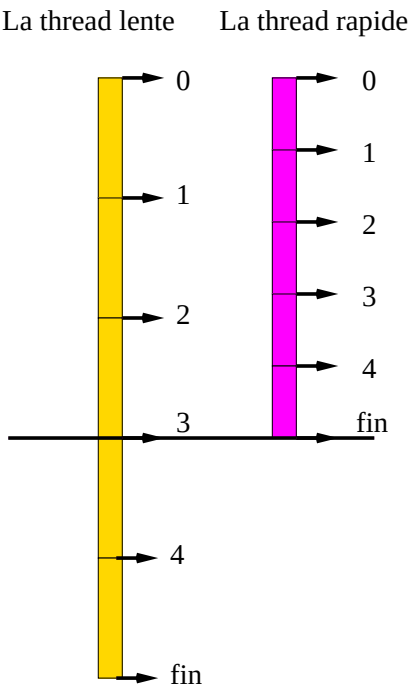
```
public class MesThread{
  public static void main(String[] args){
    Runnable tortue = new Tortue();
     Runnable lapin = new Lapin();

    Thread tortueThread = new Thread(tortue);
    Thread lapinThread = new Thread(lapin);

    tortueThread.start();
    lapinThread.start();
  }
}
```

On obtient :

```
Tortue 0 au temps 0 ms.
Lapin 0 au temps 0 ms.
Lapin 1 au temps 302 ms.
Tortue 1 au temps 515 ms.
Lapin 2 au temps 612 ms.
Lapin 3 au temps 922 ms.
Tortue 2 au temps 1025 ms.
Lapin 4 au temps 1232 ms.
Tortue 3 au temps 1535 ms.
Lapin est arrivé au temps 1542 ms.
Tortue 4 au temps 2045 ms.
Tortue est arrivée au temps 2555 ms.
```

Attention : Ici, les deux thread ont même priorité, donc même accès au processeur (équité). L'équité d'accès au processeur n'est pas assurée sur toutes les implémentations des machines virtuelles.





### Le lapin et la tortue (deuxième version)

Classe des lapins étend Thread :

```
public class Lapin extends Thread {
  public void run() {
    // inchangé
  }
}
```

Classe des tortues étend Thread :

```
public class Tortue extends Thread {
  public void run() {
    // inchangé
  }
}
```

Mise en place

```
public class MesThread{
  public static void main(String[] args){
    Thread tortueThread = new Tortue();
    Thread lapinThread = new Lapin();

    tortueThread.start();
    lapinThread.start();
  }
}
```



### Terminaison d'un processus léger

Terminaison normale : à la fin de la méthode `run()`.

On peut forcer la terminaison d'un processus léger avant la fin du `run()` en terminant l'application. L'application se termine lorsque :

- `Runtime.exit()` est appelée par l'un des processus légers;
- tous les processus légers qui n'ont pas été marqués *daemon* sont terminés.

Un processus léger peut être *user* ou *daemon*. On peut créer des processus légers *daemon* grâce à la méthode `setDaemon()` de la classe `Thread`. (Ex. `maThread.setDaemon(true);`).

*Priorités d'accès au processeur*

Les niveaux de priorité d'accès au processeur varient de 1 à 10. Des constantes de la classe `Thread` les définissent :

| | |
|---|---|
| `Thread.MAX_PRIORITY` | 10 |
| `Thread.NORM_PRIORITY` | 5 |
| `Thread.MIN_PRIORITY` | 1 |

On peut définir et consulter un niveau de priorité en appliquant une des méthodes suivantes sur l'objet de contrôle du processus léger :

- `setPriority()`
- `getPriority()`
- `setMaxPriority()`



## Exclusion mutuelle

Une *opération atomique* est une opération qui ne peut être interrompue une fois qu'elle a commencé.

- Java garantit l'atomicité de l'accès et de l'affectation des variables de type primitif (*sauf* `long` *et* `double`).
- Java possède un mécanisme d'*exclusion mutuelle* entre processus légers. Il garantit l'atomicité d'exécution de morceaux de code.

Un *verrou* peut être associé à une portion de code et permet d'exclure l'accès de deux processus légers sur cette portion.
Pour cela, on *synchronise* une portion de code relativement à un objet en utilisant le mot clé `synchronized` :

- `synchronized` comme modificateur d'une méthode : s'applique au code d'une méthode relativement à l'objet courant.
- `synchronized(obj){... portion de code ...};`

Pendant l'éxécution par un processus léger $A$ d'une portion de code `synchronized`, tout autre processus léger essayant d'exécuter une portion de code `synchronized` relative au même objet est suspendu. Une fois $A$ terminé, un seul des processus légers en attente est relancé.



### Exemple : tableau

```
public class Tableau{
  private int[] tableau;

  public synchronized int somme(){
    int s = 0;
    for (int i = 0 ; i < tableau.length ; i++)
      s += tableau[i];
    return s;
  }

  public synchronized void setElem(int i, int j){
    tableau[i] = j;
  }
}
```

Pendant l'exécution d'un `x.setElem()` ou d'un `x.somme()` dans un processus `P`, tout autre processus `Q` qui essaie de faire `x.setElem()` ou `x.somme()` sur le même `x` est suspendu.

### Exemple : variante

```
public class Tableau{
  private int[] tableau;

  public synchronized int somme(){
    int s = 0;
    for (int i = 0; i < tableau.length ; i++)
      s += tableau[i];
    return s;
  }

  public void setElem(int i, int j){
    if (i < 0 || i>= tableau.length)
      throw new IndexOutOfBoundsException();
    synchronized(this) {
      tableau[i] = j;
    }
  }
}
```

Dans cette version, seule l'affectation est vérouillée : un processus qui essaie d'écrire à un index hors bornes n'est pas suspendu.



### Sûreté et vivacité

Quelques notions :

- *sûreté (safety)* rien de faux peut se produire. L'exclusion mutuelle règle le problème de l'accès concurrent en écriture.
- *vivacité (liveness)* tout processus peut s'exécuter.

Les diverses facettes de la non vivacité :

- famine (contention): un processus léger est empêché de s'exécuter parce que un processus plus prioritaire accapare le processeur;
- endormissement (dormancy) : un processus léger est suspendu et jamais réveillé;
- terminaison prématurée;
- interblocage (deadlock) : plusieurs processus légers s'attendent mutuellement avant de continuer.



## Synchronisation entre processus légers

Java propose deux mécanismes :

- attente/notification avec `wait()` et `notify()`
  `wait()` appelé sur un objet suspend le processus léger courant qui attend une notification d'un autre processus via le moniteur de l'objet;
  `notify()` appelé sur un objet libère un processus léger en attente par `wait()` sur le moniteur du même objet.
- attente de terminaison avec `join()`
  `join()` est appelé sur l'objet de contrôle d'un processus léger dont la terminaison est attendue; le processus courant est alors interrompu jusqu'à la fin de celui-ci.

Les méthodes `wait()`, `join()` et `sleep()` peuvent être interrompues (et leur processus est alors débloqué). La méthode bloquante lève une exception `InterruptedException` qui peut être captée.



### Exemple : les tourneurs et le compteur

Cinq processus légers (les `Tourneur`) veulent faire tourner un compteur (le `Compteur`) qui compte modulo 5.
Voici le compteur:

```
public class Compteur {
  private int max; // 5 dans l'exemple
  private int count = 0; // initialisation, importante !
  public Compteur (int max) {
    this.max = max;
  }
  public int getMax(){
    return max;
  }
  public int getValue(){
    return count;
  }
  public synchronized void increment(){
    count = (count +1) % max ; // ce qu'un tourneur veut faire
  }
}
```

La règle du jeu : un Tourneur ne peut faire tourner le compteur que lorsqu'il est égal à son numéro.

Voici la classe des tourneurs.

```
public class Tourneur extends Thread{
  private Compteur c;  // le compteur
  private int numero;  // numéro du tourneur
  public Tourneur(int numero, Compteur c){
    System.out.println("Tourneur "+ numero + " créé.");
    this.numero = numero;
    this.c = c;
    if (numero + 1 < c.getMax()) {
      new Tourneur(numero + 1, c);
    }
    System.out.println("Tourneur "+ numero + " démarre.");
    start();
  }

  public void run(){ ... }

  public static void main(String[] args) {
    Compteur c = new Compteur(5);
    new Tourneur(0, c);
  }
}
```

Début d'exécution

```
Tourneur 0 créé.
Tourneur 1 créé.
Tourneur 2 créé.
Tourneur 3 créé.
Tourneur 4 créé.
Tourneur 4 démarre.
...
```



```
public void run()
  try {
    for (int etape = 0 ; ; etape++) {
      System.out.println("Tourneur "+ numero
        + " in étape "+ etape);
      synchronized(c){
        while (numero != c.getValue())
          c.wait(); // suspend this
        System.out.println("Tourneur "+ numero
          + " out étape "+ etape);
        c.increment();
        c.notifyAll();  // libère les threads supendus
      }
    }
  } catch (InterruptedException e) {}
}
```

Et le résultat:

```
...
Tourneur 4 démarre.
Tourneur 4 in étape 0
Tourneur 3 démarre.
Tourneur 3 in étape 0
Tourneur 2 démarre.
Tourneur 2 in étape 0
Tourneur 1 démarre.
Tourneur 1 in étape 0
Tourneur 0 démarre.
Tourneur 0 in étape 0
Tourneur 0 out étape 0
Tourneur 1 out étape 0
Tourneur 2 out étape 0
Tourneur 3 out étape 0
Tourneur 4 out étape 0
Tourneur 4 in étape 1
Tourneur 3 in étape 1
...
```



## Maître et esclave

Un esclave "travaille"

```
public class Esclave implements Runnable {
  private int result;
  public int getResult(){ return result; }
  public int durTravail(){ return 0; }
  public void run(){ result = durTravail(); }
}
```

Le maître fait travailler l'esclave, et attend, par `join`, la fin du processus esclave.

```
public class Maitre implements Runnable{
  public void run(){
    Esclave e = new Esclave();
    Thread esclave = new Thread(e);
    esclave.start();
    // fait autre chose
    try {
      esclave.join(); // attente de fin du run
    } catch (InterruptedException ex){}
    int result = e.getResult();
    System.out.println(result);
  }
}
```

Mise en place:

```
public class TestMaitre{
  public static void main(String[] args) {
    Maitre m = new Maitre();
    Thread maitre = new Thread(m);
    maitre.start();
  }
}
```



## Variables locales à un processus léger

On peut simuler des variables locales à chaque processus léger. Pour cela :

- on crée un objet de la classe `ThreadLocal`;
- on y accède par `Object get()`;
- on le modifie par `void set(Object o)`.

*Exemple*

```
public class MaCible implements Runnable {
    public ThreadLocal v = new ThreadLocal();
    public void run() {
      v.set(new Double(Math.random()));
      System.out.println(v.get());
    }
    public static void main(String[] args){
      MaCible c = new MaCible();
      Thread t1 = new Thread(c);
      Thread t2 = new Thread(c);
      t1.start();
      t2.start();
    }
}
```

On obtient :

```
0.8955847189505597
0.43636788900311063
```