# Aide Mémoire JAVA

*Petit guide de survie à l'attention des étudiants de Supélec.*
*Édition 2014*

*JAVA 7*

Supélec

*Cécile Hardebolle*
*Christophe Jacquet*
*Marc-Antoine Weisser*



## Déclarations de variables

Une variable stocke une valeur de *type simple* ou de *type référence* (vers un *tableau* ou un *objet*). Elle doit être déclarée et peut être initialisée dans le même temps. Plusieurs variables de même type peuvent être déclarées ensemble.

```
type nomVariable [= valeur];
type nomVar1 [=val], nomVar2 [=val] ... ;
```

*Descriptions des principaux types simples*

| Type | Description | Formes des litéraux | |
|---|---|---|---|
| `int` | Entiers signés sur 32 bits | `1, -3, -26, 2008975` | |
| `double` | Nombres à virgule flottante double précision<br>Notation scientifique possible | `123.4`<br>`-26.0` | `1.234e2`<br>`-2.6e1` |
| `boolean` | Booléen | `true, false` | |
| `char` | Caractère | `'a', 'A'` | `'\u0108'` |

| | Notations spécifiques pour certains caractères usuels | | | | | |
|---|---|---|---|---|---|---|
| Code | `'\n'` | `'\t'` | `'\\'` | `'\''` | `'\"'` | `'\b'` |
| Valeur | *À la ligne* | *Tabulation* | *Antislash* | *Apostrophe* | *Guillemets* | *Backspace* |

## Expressions et opérateurs

Une *expression* est une suite d'opérations impliquant variables, *littéraux* et *appels de méthodes*. Son type dépend des opérations et des opérandes.

*Listes des opérateurs avec les types possibles des opérandes du résultat*

| | Opérateurs | Description | Type Opérandes | Type Resultat |
|---|---|---|---|---|
| Math | `+, -, *, /` | Opérations arithmétiques | `int, double` | `int, double` |
| | `%` | Reste de la division | `int, double` | `int, double` |
| Logique | `==, !=` | Égalité, inégalité | tout | `boolean` |
| | `<, <=, >, >=` | Comparaison | `int, double, char` | `boolean` |
| | `&&, \|\|` | Et, ou (logique) | `boolean` | `boolean` |
| | `!` | Non (logique) | `boolean` | `boolean` |
| Binaire | `&, \|, ^` | Opérations bit à bit (et/ou/xor) | `int` | `int` |
| | `<<, >>, >>>` | Décalage de bit signé à gauche et à droite et non signé à droite | `int` | `int` |
| | `^` | Complément bit à bit | `int` | `int` |

## Affectation et transtypage

*'affectation* stocke le résultat de l'évaluation d'une expression dans une ariable en écrasant la précédente valeur.

```
nomVariable = expression ;      Syntaxe de l'affectation
```

e type de l'expression doit être le même que celui de la variable sinon il faut n *transtypage*, c'est-à-dire une conversion de type. Dans de rares cas, il peut tre implicite.

```
(type) expression                  Syntaxe du transtypage

int i = (int)3.14 ;                Exemple de transtypage explicite
double d = 3 ;                     Exemple de transtypage implicite
```

| Notations raccourcies pour certaines affectations fréquentes | | | | | |
|---|---|---|---|---|---|
| `var += q;` | ⇔ | `var = var + q;` | `var -= q;` | ⇔ | `var = var - q;` |
| `var *= q;` | ⇔ | `var = var * q;` | `var /= q;` | ⇔ | `var = var / q;` |
| `var++;` | ⇔ | `var = var + 1;` | `var--;` | ⇔ | `var = var - 1;` |



## Blocs d'instructions

| Description d'un bloc | Syntaxe |
|---|---|
| Groupement d'instructions entouré d'accolades dont les instructions sont *indentées*. La *portée* (l'utilisation possible) d'une variable commence à l'endroit où elle est déclarée et finit à la fin du bloc contenant sa déclaration. | `{`<br>`  instruction;`<br>`  ...`<br>`}` |

## Instructions Conditionnelles

Une instruction conditionnelle est un choix entre plusieurs blocs de code selon une condition.

| | Description des instructions conditionnelles | Syntaxe |
|---|---|---|
| `if` | Le premier bloc est exécuté si `condition` est `true`, le second sinon. Plusieurs blocs `if` peuvent être imbriqués. Le bloc `else` est optionnel. | `if(condition) {`<br>`  instruction;`<br>`  ...`<br>`} else {`<br>`  instruction;`<br>`  ...`<br>`}` |
| `switch` | L'exécution du bloc commence à l'instruction suivant le `case` dont la valeur correspond à celle de `variable` et se termine au premier `break` rencontré. Si aucun `case` ne correspond à la valeur de `variable`, c'est après le `default` que l'exécution commence. Le type de `variable` est simple ou `String`. | `switch(variable) {`<br>`  case val1:`<br>`  ...`<br>`  break;`<br>`  case val2:val3:`<br>`  ...`<br>`  default:`<br>`  ...`<br>`}` |

## Boucles

Une boucle est l'exécution répétée (*itération*) d'un bloc de code selon une condition. Le tableau suivant répertorie les quatre boucles de JAVA. Pour des exemples, voir l'affichage des éléments d'un tableau à la page suivante.

| | Description des boucles | Syntaxe |
|---|---|---|
| `while` | Exécution d'un bloc tant que `condition` est `true`. | `while(condition) {`<br>`  instruction;`<br>`  ...`<br>`}` |
| `do while` | Exécution d'un bloc une fois puis tant que `condition` est `true`. | `do {`<br>`  instruction;`<br>`  ...`<br>`} while(condition);` |
| `for` | Exécution d'un bloc tant que `cond` est `true`. En général, la condition repose sur la variable initialisée à l'entrée de la boucle (`init`) puis mise à jour à chaque itération (`maj`). | `for(init; cond; maj) {`<br>`  instruction;`<br>`  ...`<br>`}` |
| `for each` | Exécution d'un bloc pour chaque valeur contenue dans `iter` (un tableau ou objet implémentant l'interface `Iterable`). La variable `elt` prend toutes les valeurs contenues dans `iter` successivement. | `for(type elt: iter) {`<br>`  instruction;`<br>`  ...`<br>`}` |

Les instructions `continue` et `break` ne sont pas indispensables et leur utilisation n'est pas recommandée. `continue` saute les instructions restantes dans une itération et passe à la suivante. Les mises à jour de la boucle `for` sont effectuées. `break` force la sortie d'un bloc (`switch`, `for`, `while`, `do while`).

# Tableaux

## Fonctionnement Général

Un tableau contient un nombre fixe de données de même type. Une variable ne stocke jamais un tableau mais une *référence* vers un tableau. La création d'un tableau s'effectue en deux étapes : déclaration d'une variable stockant une référence puis *allocation* en mémoire d'un tableau (mot clef `new`). Deux syntaxes existent pour la déclaration.

```
type[] tab;                 Déclaration, syntaxe 1
type tab[];                 Déclaration, syntaxe 2

tab = new type[taille];     Allocation mémoire
```

Une case d'un tableau est une variable à laquelle on accède avec un *indice*. La numérotation des cases commence à 0. La valeur `length` contient le nombre de cases d'un tableau.

```
tab[0] = expression;                Affectation du premier élément
tab[tab.length-1] = expression;     Affectation du dernier élément
```

Il est possible d'allouer la mémoire et d'initialiser les éléments d'un tableau en une seule instruction avec l'une des deux notations suivantes.

```
type[] tab = new type[] { val1, val2... };
type[] tab = { val1, val2... };
```

## Tableau à plusieurs dimensions

Un tableau peut avoir plusieurs dimensions. Pour la déclaration et l'accès aux cases, la syntaxe est similaire à celle des tableaux à une dimension. Exemple pour un tableau à deux dimensions avec toutes les lignes de même taille :

```
type[][] tab = new type[nbLignes][nbColonnes];
tab[i][j] = expression;
```

Un tableau à plusieurs dimensions est un tableau de tableaux. Il est donc possible de déclarer un tableau ayant des lignes de tailles différentes. Exemple d'un tableau triangulaire :

```
type[][] tab = new type[nbLignes][];
for(int i=0; i<tab.length; i++) {
    tab[i] = new type[i+1];
}
```

## Affichage des éléments d'un tableau

Une variable ne stocke pas un tableau mais une référence dessus. L'instruction "`System.out.println( tab );`" affiche donc uniquement la référence et non les éléments du tableau. Pour afficher les éléments, on peut utiliser une boucle ou la méthode `toString` de la classe `Arrays` permettant la conversion d'un tableau en `String`.

Affichage avec une boucle for

```
String s="";
for(int i=0; i<tab.length; i++) {
    s = s+tab[i].toString()+" ";
}
System.out.println(s);
```

Affichage avec une boucle while

```
String s="";
int i=0;
while(i<tab.length) {
    s = s+tab[i].toString()+" ";
    i++;
}
System.out.println(s);
```

Affichage avec une boucle for-each

```
String s="" ;
for(Object t: tab) {
    s = s+t.toString()+" ";
}
System.out.println(s);
```

Affichage par Arrays

```
String s=Arrays.toString(tab);
System.out.println(s);
```

# Programmation Objet

## Généralité sur les classes et objets

Une classe est un moule permettant de créer des *objets* (*instances*) dont le type est une classe. Elle inclut la déclaration d'*attributs*, de *constructeurs* et de *méthodes*. Les *modificateurs* précédant le mot clef `class` sont des propriétées. La visibilité restreint l'accès et `final` rend la classe non dérivable (voir héritage).

```
[visibilité] [final] class NomClasse {
    ...

}
```

| Visibilité | Classe, attribut ou méthode accessible depuis |
|---|---|
| `private` | la classe uniquement (autorisé uniquement pour les classes de haut niveau, les attributs et les méthodes) |
| *non spécifiée* | toutes classes du package (dit «package-private») |
| `protected` | toutes classes du package et sous-classes |
| `public` | toutes classes |

## Attributs

Un attribut est une caractéristique d'un objet. Il est déclaré en début de classe.

```
[visibilité] [static] [final] type nomAttribut ;
```

| Modificateur | Propriété de l'attribut |
|---|---|
| `final` | Attribut constant |
| `static` | Un attribut `static` est associé à la classe plutôt qu'aux objets. Il a alors une valeur commune pour tous les objets de la classe. |

## Méthodes

Une méthode est un bloc d'instructions (*corps d'une méthode*) destiné à réaliser une opération. Elle peut recevoir des *paramètres* et renvoyer une *valeur de retour*. Son nom et le type de ses paramètres forment sa *signature*. Le type de la valeur de retour est `void` si la méthode ne renvoie rien.

```
[visibilité] [static] [final] type nomMethode([type param...]){
    instructions ;
}
```

| Modificateur | Propriété de la méthode |
|---|---|
| `final` | Méthode non surchargeable (voir héritage) |
| `static` | Méthode de classe. Elle ne peut utilser que des attributs `static`. |

Le mot clef `return` met fin à l'exécution de la méthode. Si une valeur de retour est attendue, elle doit être indiquée après `return`.

L'appel d'une *méthode de classe* (`static`) est préfixé par le nom de la classe.

```
NomClasse.nomMethode( val1, val2... ) ;
```

L'appel d'une *méthode d'instance* (abscence du modificateur `static`) est préfixé par un objet. Dans le corps de celle-ci, cet objet est désigné par `this`.

```
nomObjet.nomMethode( val1, val2... ) ;
```

## Méthode Main

Une classe est exécutable si elle inclut une méthode `main`. C'est le point d'entrée du programme. Les arguments donnés au programme lors de son lancement sont transmis, sous forme d'un tableau de `String`, en paramètre de la méthode `main`.

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

# Programmation Objet (suite)

## Constructeur et instanciation

Un constructeur permet d'initialiser des instances. C'est une méthode portant le nom de la classe et ne renvoyant rien (pas de `void`). Sa première instruction peut être `super(...)` pour appeler un constructeur d'une classe mère (voir héritage) ou `this(...)` pour appeler un autre constructeur de la classe.

```
[visibilité] NomClasse( type param... ) {
    [super(...)|this(...)]
    instruction;
    ...
}
```

Pour créer un objet (*instanciation*), on utilise `new` suivi du constructeur choisi pour l'initialisation. La référence sur l'objet peut être stockée dans une variable.

```
NomClasse variable = new NomClasse( param1, param2... ) ;
```

## Référence et égalité entre objets

L'opérateur d'égalité "==" teste si deux expressions sont égales. Pour les objets on teste donc si deux références désignent (physiquement) le même objet. Pour l'égalité logique, on utilise la méthode `equals` de la classe `Object`. Sa surcharge définit le mode de comparaison des instances d'une classe.

```
public boolean equals(Object obj) { ... }
```

## Exemple de classe

```
public class Complexe {
    private double rho, theta ;
    public static final Complexe UN = new Complexe(1,0);
    public static final Complexe I = new Complexe(1,Math.PI/2);

    public Complexe(double rho, double theta){
        this.rho = rho;
        this.theta = theta;
    }

    public double imag(){ return this.rho*Math.sin(this.theta);}
    public double reel(){ return this.rho*Math.cos(this.theta);}

    public Complexe division( Complexe c ){
        return new Complexe(this.rho/c.rho, this.theta-c.theta);
    }

    public String toString(){
        return "<"+this.reel()+", "+this.imag()+">";
    }

    public boolean equals(Object o){
        if( !(o instanceof Complexe) ) {
            return false;
        }

        Complexe c=(Complexe)o;
        return c.rho==this.rho && c.theta==this.theta;
    }
}

public class TestComplexe {
    public static void main( String[] args ) {
        if( !Complexe.I.equals( Complexe.UN ) ) {
            System.out.println( "1 et i sont différents" );
        }

        Complexe u = new Complexe( 1, Math.PI/6 ) ;
        System.out.println("u/i : " + u.division( Complexe.I ));
    }
}
```

# Hiérarchie de Classes

## Héritage

Une classe *hérite* d'une et une seule classe. Par défaut c'est de la classe `Object`. Il est possible d'en spécifier une autre avec `extends` :

```
class ClasseFille extends ClasseMere { ...
```

Une *classe fille* hérite de tout attribut et méthode de sa *classe mère*. L'héritage est *transitif* : attributs et méthodes se propagent à la descendance.

À l'instanciation d'une classe, les attributs hérités de la classe mère sont le plus souvent initialisés en appelant un constructeur de la classe mère. Il est désigné par le mot clef `super` suivi des paramètres. Son appel est alors la première instruction du constructeur.

```
public ClasseFille( ... ) {
    super( ... );
    instruction;
    ...
}
```

Un objet d'une classe fille a toutes les caractéristiques définies par sa classe mère, il peut donc être utilisé partout où un objet d'une classe mère est attendu.

```
ClasseMere objet = new ClasseFille( ... ) ;
```

Dans une classe fille, il est possible de *surcharger* (redéfinir) le corps d'une méthode héritée de sa classe mère. Si une méthode est surchargée, lors de son appel c'est la version définie dans la classe fille qui est utilisée.

## Test de type et transtypage

Le mot clef `instanceof` permet de tester si une expression référence un objet d'une certaine classe ou d'une de ses descendantes. Le résultat est une valeur booléenne. Le type d'une expression peut être imposé (transtypage). Attention, il doit être compatible avec l'objet référencé par l'expression.

```
objet instanceof NomClasse;              Test de type
((NonClasse)objet).nomMethode();         Transtypage
```

## Classe abstraite et interface

Une *classe abstraite* est une classe incomplète et donc non instanciable. Elle peut contenir des *méthodes abstraites*. Ce sont des méthodes dont seule la signature est définie, le corps ne l'est pas.

```
[visibilité] abstract type nomMethode( ... ) ;
```

Une classe peut hériter d'une classe abstraite. Elle restera abstraite à moins d'*implémenter* toutes les méthodes abstraites de la classe mère.

Une *interface* est une classe qui ne définit que des signatures de méthodes et des attributs constants (`final static`). Elle peut hériter d'une ou plusieurs interfaces.

```
[visibilité] interface NomInterface [extends Int1, Int2] {
    [visibilité] final static type nomAttribut ;
    ...
    [visibilité] [static] type nomMethode( ... ) ;
    ...
}
```

Une classe peut implémenter une ou plusieurs interfaces. Elle doit implémenter toutes les méthodes définies dans ses interfaces (à moins d'être abstraite).

```
class NomClasse implements NomInter1, NomInter2 { ...
```

On peut typer une variable par une interface : la variable pourra référencer tout objet implémentant l'interface et on pourra appeler les méthodes de l'interface.

# Classes Paramétrées

## Utilisation

Le *polymorphisme paramétrique* (ou *généricité*) permet de définir des types complexes (classe ou interface) paramétrés par d'autres types. Les exemples courants sont les structures de données décrites pour un type quelconque choisi lors de l'utilisation. Ainsi, la classe `ArrayList<E>` est une liste générique d'éléments de type non défini mais représenté par E, le type réel est indiqué à l'utilisation :

```
ArrayList<String> a = new ArrayList<String>();
```

## Définition

La syntaxe pour la déclaration d'une classe paramétrée par un seul type (il est possible de la paramétrer par plusieurs, exemple plus bas) est :

```
[visibilité] class NomClasse <E> { ... }
```

Au sein de la classe, il est alors possible d'utiliser les types paramètres pour la déclaration d'attributs et de méthodes. Ex. :

```
private E nomAttribut ;

public E nomMethode(ArrayList<E> uneListeParametree){...}
```

## Exemple de classe paramétrée

Classe paramétrée destinée à stocker un couple d'objets de deux types différents.

```
public class Paire<T, S> {
    private T val1;
    private S val2;

    public Paire(T val1, S val2){
        this.setValeurs( val1, val2 ) ;
    }

    public void setValeurs(T val1, S val2){
        this.val1 = val1;
        this.val2 = val2;
    }

    public T getVal1(){ return val1; }
    public S getVal2(){ return val2; }

    private void setVal1(T val1){ this.val1 = val1; }
    private void setVal2(S val2){ this.val2 = val2; }

    public boolean equals(Object o){
        if(!(o instanceof Paire<T,S>)){
            return false ;
        }
        Paire<T,S> p =(Paire<T,S>)o;
        return    p.val1.equals(this.val1)
               && p.val2.equals(this.val2);
    }

    public String toString(){
        return "<"+this.val1+","+this.val2.+">";
    }
}

public class TestPaire {

    public static void main( String args[] ){
        Paire<String, Double> unePaire ;
        unePaire = new Paire<String,Double>( "Un", 1.0) ;
    }
}
```

# API et Classes Utiles

## Structuration en package

JAVA permet de structurer les classes en ensembles appelés *packages* afin de gagner en modularité. Les classes contenues dans un fichier appartiennent au package par defaut à moins d'en indiquer un autre en première ligne du fichier.

```
package nomPackage;
```

Lorsque l'on utilise une classe donnée, le compilateur la recherche dans le package par défaut. Si elle appartient à un autre package il est nécessaire de l'importer au début du fichier. On peut importer toutes les classes d'un package avec '*'. Ceci n'importe pas les classes des sous-packages.

```
import nomPackage.NomClasse;          importe une classe seule
import nomPackage.*;                  classes de monPackage
import nomPackage.sousPackage.*;      classes de sousPackage
```

Le package `java.lang` contient l'ensemble des classes fondamentales au langage. Il est importé automatiquement.

*Packages notables de la bibliothèque standard de JAVA*

| Package | Rôle |
|---|---|
| `java.util` | Structures de données |
| `java.io` | Opérations d'entrée/sortie de base |
| `java.math` | Opérations sur des nombres à précisions arbitraire |
| `java.nio` | Opérations d'entrée/sortie haute performance |
| `java.net` | Opérations réseau, sockets, ... |
| `java.security` | Cryptographie |
| `java.awt` | Primitives graphiques |
| `javax.swing` | Package pour GUI indépendante de la plateforme |

## java.lang.String

Cette classe représente les chaînes de caractères. Tout texte entre guillemets est une chaîne. Une chaîne peut contenir des caractères spéciaux (Ex. `'\n'`). On concatène deux chaînes avec "+".

```
String s = "une chaine" + "une autre\n" ;
```

Une chaîne peut être concaténée avec une valeur de type simple. La valeur est automatiquement convertie en `String`. Une chaîne peut également être concaténée avec un objet. Dans ce cas, l'objet est converti en `String` par appel implicite à la méthode "`public String toString( )`". Cette dernière est définie dans la classe `Object` et peut être surchargée.

Une chaîne n'est pas modifiable. Une variable peut donc référencer un nouvel objet `String` mais pas le modifier en soi.

## java.lang.Math et java.util.Arrays

Les classes `Math` et `Arrays` regroupent des constantes et des méthodes de classe pour les mathématiques et la manipulation de tableaux.

| | Méthode | Description |
|---|---|---|
| Math | `abs, sqrt, pow` | Valeur absolue, racine carré et puissance |
| | `sin, cos, tan` | Fonctions trigonométriques |
| | `log, log10, exp` | Logarithme népérien, en base 10 et exponentiel |
| | `floor, ceil` | Partie entière inférieure et supérieure d'une valeur |
| | `min, max` | Minimum et maximum de deux valeurs |
| Arrays | `toString()` | Conversion d'un tableau en chaîne de caractères |
| | `sort(type[])` | Trie d'un tableau par ordre croissant des valeurs |
| | `binarySearch(type[], type)` | Renvoie l'indice d'une valeur dans un tableau trié |
| | `copyOf(type[])` | Renvoie une copie d'un tableau |

# Structures de Données Avancées

## Collection, List et Set

L'interface `java.util.Collection<E>` regroupe les méthodes génériques pour la gestion de structures de données. Elle hérite de `Iterable<E>`. Les structures l'implémentant peuvent être parcourues avec une boucle *for each*.

Les interfaces List<E> et Set<E> héritent de Collection<E>. Les classes qui les implémentent sont respectivement des listes et des ensembles.

*Liste (non exaustive) des méthodes définies dans Collection et List*

| | Méthode | Description |
|---|---|---|
| Collection<E> | `boolean add(E e)`<br>`boolean addAll(Collection c)` | Insère `e` (ou les élements de `c`) et renvoie `true` si la structure a été modifiée |
| | `void clear( )` | Vide la `Collection` |
| | `boolean contains(Object o)`<br>`boolean containsAll(Collection c)` | Renvoie `true` si `o` ou l'ensemble des éléments de `c` sont dans la structure |
| | `boolean isEmpty( )` | Renvoie `true` si la liste est vide |
| | `boolean remove(Object o)`<br>`boolean removeAll(Collection<E> c)` | Supprime une occurrence de `o` ou de chaque élément de `c`. Renvoie `true` si chaque occurrence a été supprimée. |
| | `int size()` | Renvoie le nombre d'éléments |
| | `Object[] toArray()` | Renvoie un tableau des éléments |
| List<E> | `boolean add(E e)` | Insère `e` en fin de liste. Renvoie `true` |
| | `void add(int i, E e)` | Insère `e` à l'indice `i` |
| | `E get(int i)` | Renvoie l'élément contenu à l'indice `i` |
| | `boolean remove(int i)` | Supprime l'élément à l'indice `i` |
| | `List<E> subList(int from, int to)` | Renvoie la sous liste entre `from` et `to` |
| Set<E> | `boolean add(E e)` | Insère `e` s'il n'existe pas d'élément `f` vérifiant `e.quals(f)`. Renvoie `true` si le `Set` a changé. |

`ArrayList<E>` est une implémentation de `List<E>` construite sur des tableaux. L'accès aux éléments est rapide à partir de leur indice. Suppression et Insertion sont plus coûteuses.

`HashSet<E>` est une implémentation de `Set<E>` construite sur une table de hachage. Accès, insertion et suppression sont en temps constant.

## Map

L'interface `java.util.Map<K,V>` est utilisée pour les tableaux d'association clef-valeur (*dictionnaires*). On insère dans la table une valeur en l'associant à une clef. Une clef ne peut être associée qu'à une seule valeur. Pour retrouver une valeur on utilise la clef. La principale implémentation est HashMap<K,V>.

*Liste de certaines méthodes définies dans Map*

| Méthode | Description |
|---|---|
| `void clear()` | Supprime toutes les associations d'une table |
| `boolean containsKey(Object k)` | Renvoie `true` si la table contient la clef `k` |
| `boolean containsValue(Object v)` | Renvoie `true` si la table contient la valeur `v` |
| `V get(Object k)` | Renvoie la valeur associé à la clef `k` ou `null` |
| `boolean isEmpty( )` | Renvoie `true` si la liste est vide |
| `Set<K> keySet()` | Renvoie un ensemble contenant les clefs |
| `V put(K k, V v)` | Insère l'association de la clef `k` et de la valeur `v`. Renvoie la précédente valeur associée à `k` ou `null`. |
| `V remove(Object k)` | Supprime l'association avec la clef `k`. Renvoie la valeur qui lui était associée ou `null` |
| `int size()` | Renvoie le nombre d'associations |
| `Collection<V> values()` | Renvoie une collection des valeurs |

# Exceptions

## Attraper les exceptions

Les exceptions sont des objets héritant de `throwable` et utililisées pour gérer les cas d'erreurs d'un programme. Un bloc protégé peut *attraper* les exceptions *lancées* (déclenchées) par les instructions du bloc et les gérer.

| Description | Syntaxe des exceptions |
|---|---|
| `try{...}` : code protégé<br>`catch(Excep1 e){...}` : si le bloc protégé lève une exception de type `Excep1`, elle est stockée dans `e` et le bloc associé est exécuté.<br>`catch(Excep2\|...\|ExcepN e){...}` : idem, plusieurs types d'exception.<br>`finally {...}` : bloc de code exécuté quoi qu'il arrive (exception ou non). | `try {`<br>`  instructions;`<br>`  ...`<br>`} catch(Excep1 e) {`<br>`  instructions;`<br>`  ...`<br>`} catch(Excep2\|...\|ExcepN e) {`<br>`  instructions;`<br>`  ...`<br>`} finally {`<br>`  ...`<br>`}` |

Certains objets (comme les flux) doivent être refermés après leur utilisation. Lever une exception peut interompre le code qui était destiné à les refermer. Il est possible de refermer automatiquement les objets implémentant `AutoCloseable` en les déclarant entre paranthèses juste après `try`.

```
try(AutoCloseableClass a=...) {
   ...
} catch ...
```

Plutôt que traiter localement une exception, il est possible de déléguer cette tâche à la méthode appelant la méthode courante. On déclare la méthode courante comme succeptible de lever certains types d'exceptions. Si le cas se présente, l'exception levée est relayée à la méthode appelante. La méthode courante prend fin.

```
type methode(...) throws Excep1, Excep2, ... { ... }
```

## Créer des exceptions

Une exception est un objet d'une classe héritant de `Throwable`. Lancer une exception se fait avec le mot clef `throw`.

```
throw new MonException( ) ;
```

Lorsqu'une méthode succeptible de lancer une exception n'est pas dans un bloc protégé, le compilateur engendre une erreur. Dans le cas des exceptions héritant de `Error` ou `RunTimeException`, le bloc n'est pas nécessaire.

## Exceptions courantes

| Exception | Cause |
|---|---|
| `NullPointerException` | Accès à une méthode ou à un attribut depuis une variable référençant `null`. |
| `IndexOutOfBoundException` | Accès à une case n'existant pas dans un tableau. |
| `ClassCastException` | Transtypage d'un objet dans un type auquel il n'appartient pas. |

## Quitter un programme

La méthode `exit` permet de terminer l'exécution d'un programme. Elle prend en paramètre un statut qui sera renvoyé au système d'exploitation. Par convention, toute valeur non nulle indique une terminaison anormale.

```
System.exit(status)
```

# Commentaires et Javadoc

## Commentaires

Deux syntaxes sont possibles pour les commentaires. La première permet des commentaire sur une ligne, la seconde sur plusieurs lignes.

```
// Commentaire d'une ligne

/* Commentaire de plusieurs
      lignes... */
```

## Javadoc

*Javadoc* est un programme qui permet d'extraire la documentation d'une bibliothèque (API) depuis les commentaires d'un programme. Il est nécessaire d'utiliser une syntaxe précise dans les commentaires.

Le fonctionnement est le suivant : on fait précéder chaque déclaration de classe, d'attribut et de méthode d'un commentaire sur plusieurs lignes débutant par "`/**`". Ce commentaire peut inclure des balises donnant des précisions.

```
/**    Constructeur d'un nombre complexe
       @param rho valeur du module
       @param theta valeur de l'angle (en radian)
*/
public Complexe(double rho, double theta){
       this.rho = rho;
       this.theta = theta;
}
```

*Liste des balises Javadoc*

| Tags | Description |
|---|---|
| `@param nom description` | Description d'un paramètre d'une méthode |
| `@return description` | Description de la valeur de retour d'une méthode |
| `@throws type` | Indique les types d'exceptions pouvant être levés |
| `@author nom` | Nom de l'auteur |
| `@version numéro` | Numero de la version |
| `@see reference` | Renvoie vers une méthode, classe, ... |
| `@since date` | Date d'apparition |
| `@deprecated commentaire` | Indique de ne plus utiliser cet élément et un commentaire |

# À Propos

Memento JAVA, version 3, septembre 2014.

Ce memento est un outil d'aide à la programmation. Il tente de regrouper la syntaxe des principaux concepts de JAVA. Il n'a pas vocation à remplacer un cours.

La syntaxe utilisée pour décrire les éléments du langage est la suivante :

- Le code utilise une police particulière : `System.exit(0);`
- Les mots clefs de JAVA sont en rouge : `int`, `void`, `for`, `public`.
- Les élements optionels sont entourés de grande paranthèses : [`abstract`]
- Les éléments en italique représentent des noms dépendant du programmeur : *`nomVariable`*, *`nomClasse`*, *`instruction`*...

*Merci à tous les relecteurs.*