# Java

Michaël Mathieu

21 novembre 2018

## Table des matières

# 1 Installation de l'environnement de développement

## 1.1 JDK – Java Development Kit

Télécharger la dernière version du JDK : `http://www.oracle.com/technetwork/java/javase/downloads`

Créer une variable d'environement système *JAVA_HOME* qui contient le chemin du répertoire où se trouve *java*. Exemple : `C:\Program Files\Java\jdk1.8.0_102`

Ajouter le répertoire *bin* de cette variable au *Path*.

Vérifier que Java est correctement installé en exécutant la commande `java -version`

## 1.2 IDE – Eclipse

Télécharger la dernière version d'Eclipse : `https://www.eclipse.org/downloads/eclipse-packages` ⇒ Eclipse IDE for Java Developers

Unzipper le répertoire sur le disque C :

Démarrer Eclipse

# 2 Matériel pédagogique

Des flashcards vous permettrons de réviser un sujet. Ces dernières sont disponibles ici : `http://sybextestbanks.wiley.com/course/index/id/60`

Afin de pouvoir vous créer un compte, il faudra que vous répondiez à une question qui se trouve dans le livre. Votre professeur vous l'indiquera.

Les flashcards se trouvent dans l'onglet *Other study tools*.

Recommandation : Par expérience, notre cerveau enregistre facilement les bonnes réponses et les tests de connaissances se tranforment rapidement en tests de mémoire. Il est recommandé de n'utiliser ce matériel que lorsqu'on est prêt ou que l'on a terminé un chapitre.

# 3 Java Building Blocks

## 3.1 Commentaires

```
/*
 *  // anteater
 */
// bear
// // cat
// /* dog */
/* elephant */
/*
 * /* ferret */
 */
```

Tous les commentaires sont OK, sauf *ferret* qui possède un `*/` de trop.

### 3.2 Classes vs. Fichiers

Chaque fichier *.java doit avoir une classe publique qui correspond au nom du fichier. Dans un fichier, il y a au maximum une classe publique. Mais il est possible d'avoir d'autres classes (visibilité protégée, paquet, privée).

### 3.3 Imports

Pour utiliser des classes qui ne sont pas dans le même paquet que la classe courante, il faut les importer. Exception : toutes les classes du paquet `java.lang` sont importées par défaut. Ainsi, il est inutile d'importer `java.lang.System`.
Pour importer toutes les classes d'un paquet il suffit de mettre le nom du paquet suivi d'un astérisque, exemple : `import java.util.*;`
Il n'est pas possible d'avoir plus d'un astérisque dans la déclaration d'un import.
ATTENTION ! Des classes ayant le même nom peuvent se trouver dans différents paquets. Par exemple, `Date` se trouve dans les paquets `java.util` et `java.sql`.

Lorsqu'Eclipse organise les imports, il ne laisse aucun astérisque, mais explicite chaque classe importées.

### 3.4 Constructeurs

Si aucun constructeur n'est définit, un constructeur par défaut sans argument est généré. Du moment que nous créons un constructeur, alors le constructeur par défaut n'est pas généré.

Il est possible de définir un bloc d'initialisation, ce dernier est exécuté **avant** le constructeur.

### 3.5 Types

Il existe les types primitifs par exemple `int` et les types objets (ou référencés) par exemple `Integer`. Dans le cas d'un type objet, nous avons à faire à un objet et possède donc des méthodes utilitaires et non uniquement une valeur.
ATTENTION ! Pour comparer des types primitifs, nous pouvons utiliser le `==`, mais pour des types objets nous devons utiliser la méthode `equals`.
Si nous comparons un type objet et un type primitif avec le `==`, le type objet est converti en type primitif (unboxing), il se peut que cette conversion génère une `NullPointerException` si le type objet est `null`. Si nous comparons un type objet et un type primitif avec la méthode `equals`, le type primitif est converti en type objet (autoboxing).

Si nous affectons un type objet à un type primitif, la valeur du type objet est convertie en type primitif (unboxing).
Si nous affectons un type primitif à un type objet, la valeur du type primitif est convertie en type objet (autoboxing).

### 3.6 Identifiants

Les identifiants (s'applique à tout) doivent : commencer par une lettre, un $, ou un _
Il est ensuite possible de mettre des chiffres.

Evidemment, il n'est pas possible d'utiliser des mots réservés du langage.

**Conventions de nommage**
- les méthodes et la variables commencent par une minusucule et utilisent le CamelCase
- les classes commencent par une majuscule et utilisent le CamelCase

### 3.7 Variables

Les variables locales n'ont pas de valeur par défaut et doivent avoir été initialisées avant d'être utilisées.
Si l'initialisation est liée à une condition, elle doit être faite dans chaque branche du code.

Les attributs d'une classe sont initialisés par défaut :

| | |
|---|---|
| `boolean` | false |
| `byte, short, int, long` | 0 |
| `float, double` | 0.0 |
| `char` | `'\u0000'` (NUL) |
| n'importe que objet | `null` |

### 3.8 Elements dans une Classe

| **Elément** | **Exemple** | **Obligatoire ?** | **Emplacement** |
|---|---|---|---|
| Déclaration de paquet | `package abc;` | Non | Première ligne du fichier |
| Imports | `import java.util.*;` | Non | Juste après le paquet |
| Déclaration de la classe | `public class C` | Oui | Juste après les imports |
| Déclaration des champs | `int value;` | Non | N'importe où dans la classe |
| Déclaration de méthodes | `void method()` | Non | N'importe où dans la classe |

## 4 Class Design

### 4.1 Héritage

Les buts de l'héritage en Programmation Orientée–Objet sont multiples. Nous pouvons notamment citer la réutilisabilité (en factorisant le code), une meilleure lisibilité (les classes ne contiennent que ce qu'il faut et qui est pertinent dans leur contexte), le polymorphisme, la liaison dynamique, etc.

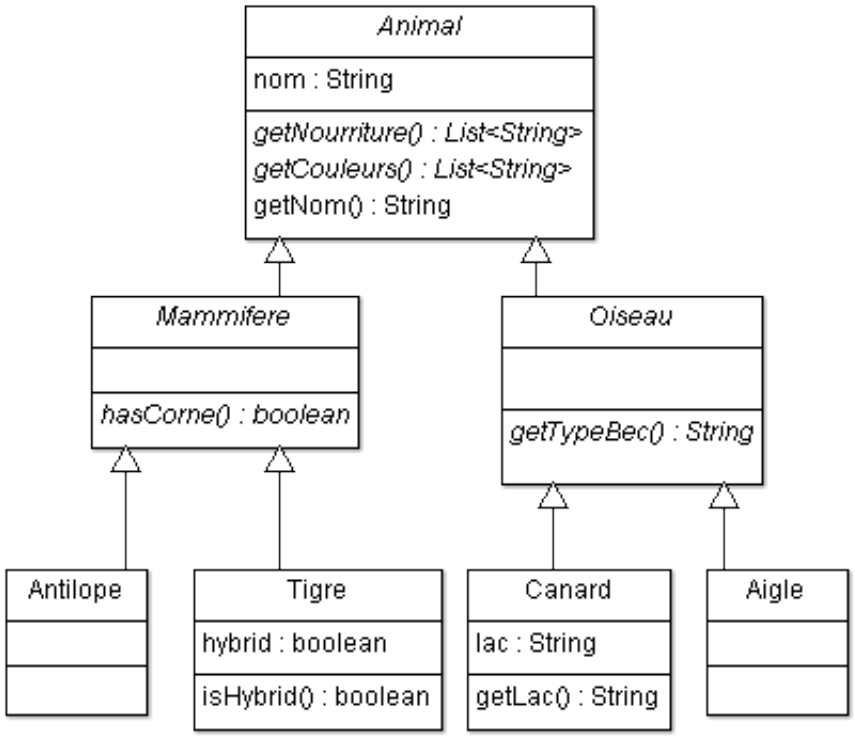


FIGURE 1 – Héritage simple

Pour le diagramme de classe UML ci–dessus, le code source correspondant est :

Animal.java

```
public abstract class Animal {
  public String nom;

  public abstract List<String> getNourriture();

  public abstract List<String> getCouleurs();

  public String getNom() {
    return nom;
  }
}
```

Mammifere.java

```
public abstract class Mammifere extends Animal {
  public abstract boolean hasCorne();
}
```

Oiseau.java

```
public abstract class Oiseau extends Animal {
  public abstract String getTypeBec();
}
```

Antilope.java

```
public class Antilope extends Mammifere {
}
```

Tigre.java

```
public class Tigre extends Mammifere {
  public boolean hybrid;

  public boolean isHybrid() {
    return ...;
  }
}
```

Canard.java

```
public class Canard extends Oiseau {
  public String lac;

  public String getLac() {
    return ...;
  }
}
```

Aigle.java

```
public class Aigle extends Oiseau {
}
```

## 4.2 Constructeurs

Une classe fille **doit** de manière implicite ou explicite utiliser un des constructeurs de la classe mère.

Par *implicite*, nous entendons l'utilisation du constructeur de la classe mère sans paramètre.

Par *explicite*, nous entendons l'appel à un des constructeurs de la classes mère (première instruction du constructeur de la classe fille). Le constructeur de la classe mère que la classe fille invoque, peut-être le constructeur sans paramètre (`super();`) ou un autre le cas échéant, par exemple `super("Tigrou");`.

## 4.3 Polymorphisme

Si nous implémentons une méthode qui affiche le nom de n'importe quel animal, nous ne souhaitons pas écrire une méthode par animal (càd une méthode pour les tigres, une pour les aigles, etc.). Etant donné que le traitement est identique pour tous les animaux (car ils possèdent tous un nom), et qu'ils héritent tous directement ou indirectement de la classe *Animal*, nous pouvons écrire cette méthode ainsi :

```
public static void printAnimal(final Animal animal) {
  System.out.println(animal.getNom());
}
```

## 4.4 Liaison dynamique

Le concept de liaison dynamique est indissociable des notions de redéfinition (override) et de méthode.
Prenons l'exemple de la méthode `public String toString()` qui est appellée notamment lorsque l'on souhaite afficher un objet. Cette méthode est définie dans la super classe de tout objet en Java, la classe `Object`. Lorsque pour une classe donnée on souhaite modifier le comportement de cette méthode nous *redéfinissons* cette méthode (signature identique) avec le code que nous souhaitons. Java indique un avertissement si nous ne préfixons pas la méthode redéfinie avec l'annotation `@Override`. Cette dernière a pour but d'informer le développeur du mécanisme qu'il est en train de mettre en place.

Si pour la classe *Tigre*, nous souhaitons lorsqu'une instance de cette classe est affichée indiquer s'il s'agit d'un tigre hybride ou non, nous devons redéfinire la méthode `toString()`.
La *liaison dynamique* est le mécanisme de Java qui va appeler la méthode la plus spécifique possible. Par exemple, si nous redéfinissons la méthode `toString()` pour la classe *Animal* et *Mammifere*, avec le programme suivant (Test.java), nous afficherions `[MAMMIFERE] Georgette`

Animal.java

```
...
@Override
public String toString() {
  return "[ANIMAL] " + getNom();
}
...
```

Mammifere.java

```
...
@Override
public String toString() {
```

```
  return "[MAMMIFERE] " + getNom();
}
...
```

Test.java

```
...
public static void main(String[] args) {
  Antilope georgette = new Antilope("Georgette");
  System.out.println(georgette);
}
...
```

### 4.5 Interface

Une interface représente un contrat qui doit être respecté par les classes l'implémentant.
Contrairement à l'héritage, une classe peut implémenter plusieurs interfaces.
Les interfaces permettent de tirer plus pleinement profit du mécanisme de polymorphisme.
Le mécanisme d'héritage fonctionne également avec les interfaces.

Une interface est par définition `abstract`, sa visibilité est `public` ou *package*.
Les attributs d'une interface sont par définition `public`, `static`, `final` comme une constante.
Nous la nommerons donc tout en majuscules avec des *underscore* pour séparer les termes.
Les méthodes d'une interface sont par définition `public`, `abstract`.

Exemple :

```
public interface Nageur {
  int VITESSE_MINIMUM_EN_KMH = 1;

  int getVitesseNageKmh();
}
```

Nous aurions pu écrire la même interface de cette manière, mais cela n'est pas recommandé :

```
public abstract interface Nageur {
  public static final int VITESSE_MINIMUM_EN_KMH = 1;

  public abstract int getVitesseNageKmh();
}
```

Propriétés d'une interface :

- il n'est pas possible d'instancier directement une interface (sauf si on fait une déclaration anonyme ; cette notion dépasse le cadre de ce cours)
- une interface peut être vide (elle n'a pas besoin d'avoir des attributs ou des méthodes)
- une interface ne peut être marquée comme `final`
- les méthodes/attributs d'une interface ne peuvent pas être marqué `private` ou `protected`, car par défaut ils sont `public`
- les méthodes d'une interface ne peuvent pas être marquées comme `final`, car par défaut elles sont `abstract`

A NOTER : si une classe est marquée comme *abstraite*, il n'est pas obligatoire d'implémenter les méthodes des interfaces qu'elle implémente.

**Implémentation par défaut** Il est possible de spécifier une implémentation par *défaut* pour les méthodes d'une interface. Ainsi, seules les classes qui souhaitent modifier ce comportement par défaut devront redéfinir ces méthodes.

Exemple :

```
public interface PossedeNageoires {
  default int getNbNageoires() {
    return 4;
  }
}

public interface Requin extends PossedeNageoires {
  default int getNbNageoires() {
    return 8;
  }
}
```

**Méthodes `static`** Il est possible de définir des méthodes `static`, mais ces dernières ne sont jamais héritées. Cela signifie que depuis une classe qui implémente une interface avec une méthode `static`, il faut préfixer le nom de la méthode avec le nom de l'interface en question.

# 5 Java API

## 5.1 String

### 5.1.1 Concaténation

Permet de joindre plusieurs chaînes de caractères ensemble. Pour la concaténation, l'opérateur utilisé est le +. Voici les 3 règles qui s'appliquent avec l'opérateur + :

1. Si les 2 opérandes sont numériques, le + signifie une addition
2. Si un des opérandes est un `String`, le + signifie une concaténation
3. L'expression est évaluée de gauche à droite

Exemples :

```
System.out.println(1 + 2);             // 3
System.out.println("a" + "b");         // ab
System.out.println("a" + "b" + 1 + 2); // ab12
System.out.println(1 + 2 + "a" + "b"); // 3ab
```

L'opérateur += peut également être utilisé pour les variables de type `String` :

```
String s = "1";        // s contient "1"
s += "2";              // s contient "12"
s += "3";              // s contient "123"
System.out.println(s); // 123
```

### 5.1.2 Immutabilité

Un objet de type `String` est immutable. Cela signifie que si on modifie sa valeur, c'est un nouvel objet qui sera créé. Ce comportement peut avoir des incidences sur la mémoire.

### 5.1.3 The String Pool

Le pool de `String` permet à Java d'éviter d'avoir plusieurs instances de la même chaîne de caractères en mémoire. Malheureusement, ce mécanisme n'est pas utilisé dans tous les cas et n'est pas très rapide.

Par exemple, le code suivant va créer 3 `String` avec la même valeur :

```
String name = "Java";
String nameBis = new String("Java");
String nameTer = "Ja";
nameTer += "va";
```

Dans l'exemple précédent le problème est que nous avons appelé directement le constructeur ou que nous avons eu recours à la concaténation au *runtime* et non à la *compilation*. Dans l'exemple suivant, un seul objet `String` sera créé, car la concaténation peut avoir lieu à la *compilation* :

```
String name = "Java";
String nameBis = "Ja" + "va";
```

Malheureusement lors de la *désérialisation*, cette optimisation n'est pas appliquée, sauf si le programme envoyant les données l'a fait.

Si vous avez un `String` et que vous souhaitez forcer Java à utiliser l'objet équivalent qui provient de son pool (Java va peut-être devoir le créer !), vous pouvez écrire :

```
String name = "Java";
String nameBis = new String("Java");

System.out.println(name == nameBis); // false

nameBis = nameBis.intern();
System.out.println(name == nameBis); // true
```

ATTENTION ! Cette instruction n'est pas très rapide et complexifie votre code. Il faut donc l'utiliser uniquement si vous savez que vous risquez d'avoir plusieurs instances en mémoire pour la même chaîne de caractères. Par exemple, lorsque l'on utilise *Hibernate* comme fournisseur de JPA et que nous avons une structure de données dénormalisée.

**StringBuilder** Lorsque nous concaténons des `String`, le `StringBuilder` permet d'optimiser le processus tant en terme de mémoire que de temps. Son utilisation est assez simple :

```
public String getStatut(final String temps) {
    StringBuilder builder = new StringBuilder();

    builder.append("Aujourd'hui nous sommes le ")
    .append(new Date())
    .append(" et le temps est ")
    .append(temps);

    return builder.toString();
}
```

### 5.1.4 Méthodes utiles

La liste exhaustive, ainsi que les différents paramètres pour chaque méthode se trouvent ici : `http://docs.oracle.com/javase/8/docs/api/java/lang/String.html`

- `length()` : le nombre de caractères dans un `String`
- `charAt()` : le caractère à une position donnée allant de 0 à `length()-1`
- `indexOf()` : l'index pour un caractère/chaîne de caractères
- `substring()` : une sous chaîne de caractères à partir d'un `String`
- `toLowerCase()` et `toUpperCase()` : pour mettre toute la chaîne de caractères en minuscule/majuscule
- `equals()` et `equalsIgnoreCase()` : pour comparer deux chaînes de caractères
- `startsWith()` et `endsWith()` : indique si un `String` commence/termine avec une certaine chaîne de caractères
- `contains()` : indique si une chaîne de caractères est contenue dans une autre
- `replace()` : pour remplacer une chaîne de caractères par une autre
- `trim()` : pour supprimer les espaces en début et en fin

## 5.2 Arrays

Les tableaux permettent de stocker plusieurs valeurs du même type et elles sont identifiées par un index allant de 0 à *taille du table – 1*.

Dans l'exemple suivant, nous créons un tableau d'entiers pour 3 éléments :

```
int[] numbers = new int[3];
```

Ces entiers peuvent être utilisés comme une variable classique :

```
numbers[0] = 1;
numbers[1] = 10;
numbers[2] = 100;
```

Il est possible de déclarer un tableau de 4 manières différentes, mais dans la pratique, nous n'utilisons que la première :

```
int[] numbers;
int [] numbers;
int numbers[];
int numbers [];
```

Si nous écrivons `int[] ids, types` nous déclarons 2 tableaux d'entiers.

Si nous écrivons `int ids[], types` nous déclarons 1 tableau d'entiers (`ids`) et un entier (`types`).

Il est possible de déclarer un tableau et de spécifier à ce moment son contenu :

```
int[] numbers = {1, 10, 100};
```

La taille du tableau est alors déduite et dans ce cas, elle sera de 3.

### 5.2.1 length

La taille d'un tableau peut être obtenu avec l'attribut `length`. Cet attribut est utilisé la plupart du temps lorsque l'on souhaite itérer sur tous les éléments d'un tableau :

```
int[] numbers = {1, 10, 100, 2000};
for (int i = 0; i < numbers.length; i++) {
  System.out.println(numbers[i]);
}
```

### 5.2.2 Sorting

Pour trier un tableau, Java met à disposition une méthode de tri dans la classe `Arrays`.

Cette méthode peut s'utiliser ainsi :

```
int[] numbers = {5, 4, 3, 2, 1};
Arrays.sort(numbers);
// numbers = {1, 2, 3, 4, 5}
```

### 5.2.3 Searching

Java fournit également une méthode permettant de rechercher l'index d'un élément dans un tableau, mais uniquement si le tableau a été trié préalablement. Si l'élément n'est pas trouvé, une valeur négative sera retourné et indiquera la position où cet élément doit être ajouté pour conserver l'ordre – 1 :

```
int[] numbers = {2, 4, 6, 8};
System.out.println(Arrays.binarySearch(numbers, 2)); // 0
System.out.println(Arrays.binarySearch(numbers, 4)); // 1
System.out.println(Arrays.binarySearch(numbers, 1)); // -1
System.out.println(Arrays.binarySearch(numbers, 3)); // -2
System.out.println(Arrays.binarySearch(numbers, 9)); // -5
```

### 5.2.4 Varargs

Un nombre d'arguments variables comme paramètres d'une méthode est converti en un tableau.

```
public void method(int number, String... names) {
  for (int i = 0; i < names.length; i++) {
    System.out.println(i + " => " + names[i]);
  }
}
```

Dans l'exemple précédent, nous voyons que le tableau `names` contient les arguments de taille variable.
Voici des exemples d'appels à cette méthode :

```
method(0);                // le tableau est vide
method(0, "a");           // le tableau contient uniquement "a"
method(0, "a", "b", "c"); // le tableau contient "a", "b", "c"
```

Deux règles s'appliquent à ce mécanisme :

1. On ne peut avoir qu'un paramètre avec un nombre d'arguments variables par signature de méthode
2. Le paramètre avec un nombre d'arguments variables, doit être placé à la fin des paramètres d'une méthode

### 5.2.5 Multidimensional Arrays

Un tableau peut avoir plusieurs dimensions afin de stocker des informations comme un carré, un cube, etc. Chaque dimension peut avoir une taille différente.

```
int[] oneDimension;
int[][] twoDimensions;     // carré
int[][][] threeDimensions; // cube
```

Voici un exemple d'utilisaition d'une tableau à deux dimensions :

```
String[][] twoDimensions = new String[3][2];
for (int i = 0; i < twoDimensions.length; i++) {
  for (int j = 0; j < twoDimensions[i].length; j++) {
    twoDimensions[i][j] = i + ";" + j;
    System.out.print(twoDimensions[i][j] + " ");
  }
  System.out.println();
}
```

```
0;0 0;1
1;0 1;1
2;0 2;1
```

## 5.3 ArrayList

En Java, il existe deux majeures implémentations de l'interface `java.util.List` : `LinkedList` et `ArrayList`.

La différence fondamentale entre ces deux implémentations est illustrée sur l'image ci–dessous : la `LinkedList` est une liste chaînée qui est très efficace pour ajouter des éléments, mais qui est moins performante pour obtenir un élément dans la liste.
Dans le cas de l'`ArrayList`, c'est l'inverse : l'ajout d'éléments peut être moins performant si le tableau sous–jacent doit être redimensionné, mais pour obtenir un élément cela est très performant, car les éléments sont stockés comme un tableau.

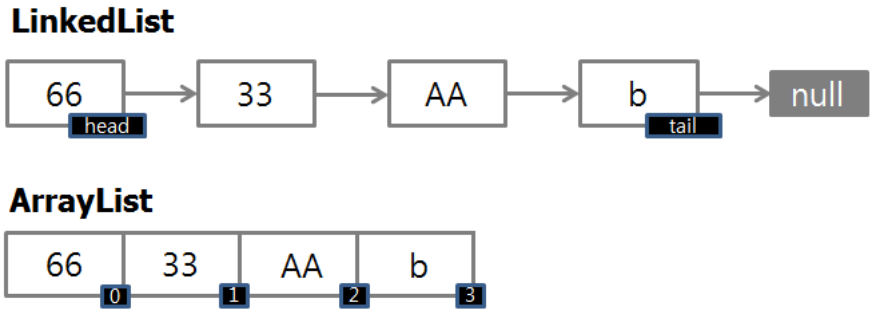


FIGURE 2 – `LinkedList` vs. `ArrayList`

Dans ce cours, nous allons nous concentrer sur l'implémentation de l'`ArrayList`.

Il y a trois possibilités pour créer une `ArrayList` :

1. `new ArrayList()` : crée une liste vide d'une capacité arbitraire de 10
2. `new ArrayList(100)` : crée une liste vide d'une capacité de 100
3. `new ArrayList(list2)` : crée une liste en copiant une autre (copie uniquement la structure : les objets ne sont pas clonés)

ATTENTION ! Ne confondez pas *taille* d'une liste et *capacité* d'une liste. Nous avons vu qu'il est possible de créer une liste avec une *capacité* de 10, mais tant que cette liste est vide sa *taille* sera de 0 ; de 1 après avoir ajouté 1 élément, etc.

### 5.3.1 Méthodes utiles

La liste exhaustive, ainsi que les différents paramètres pour chaque méthode se trouvent ici : `http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html`

- `add()` : ajouter un élément à la fin de la liste
- `remove()` : permet de supprimer de la liste un élément, soit basé sur son index, soit sur l'élément lui–même
- `set()` : permet de modifier un élément à une position donnée (ne change pas la *taille* de la liste)
- `isEmpty()` : indique si la liste est vide ou non
- `size()` : retourne la *taille* de la liste et non la *capacité*
- `clear()` : vide la liste de tous les éléments (la *capacité* reste identique)
- `contains()` : indique si la liste contient un élément
- `equals()` : compare deux listes : si elles contiennent les mêmes éléments dans le même ordre
- `toArray()` : permet de convertir une liste en tableau
- `Arrays.asList()` : convertit un tableau en liste
- `Collections.sort()` : trie une liste

## 5.4 Dates and Times

Dans les versions précédentes de Java, il existait qu'un seul objet pour stocker les dates, les heures et les timestamps : la classe `java.util.Date`.

Avec la version 8 de Java, 3 objets spécifiques ont été créés :
- `LocalDate` : contient uniquement la date
- `LocalTime` : contient uniquement l'heure
- `LocalDateTime` : contient une date et une heure

### 5.4.1 Méthodes utiles

La liste exhaustive, ainsi que les différents paramètres pour chaque méthode se trouvent ici : `http://docs.oracle.com/javase/8/docs/api/java/time/LocalDate.html`
- `of()` : permet de créer une date (méthode statique)
- `plus...(), minus...()` : permet d'ajouter/enlever $n$ seconde(s), minute(s), heure(s), jour(s), mois, année(s) à une date/heure

### 5.4.2 Formatting

Sur une date, il est possible d'obtenir le jour du mois (`getDayOfMonth()`), le mois (`getMonth()`), l'année (`getYear()`).

Le formatage permet d'obtenir un objet de type date sous la forme d'une chaîne de caractères à afficher.
A cet effet, vous pouvez utiliser les formateurs disponibles dans la classe `java.time.format.DateTimeFormatter` : `ISO_LOCAL_DATE`, `ISO_LOCAL_DATE_TIME`, etc. ou utiliser votre propre pattern : `DateTimeFormatter.ofPattern("dd.MM.yyyy")`.
Les différentes règles régissant ces patterns se trouvent ici : `http://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html`.

### 5.4.3 Parsing

Le parsing est l'inverse du formatage : permettre d'obtenir un objet de type date à partir d'une chaîne de caractères.

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("dd.MM.yyyy");
LocalDate date = LocalDate.parse("12.08.1982", formatter);
```

# 6 Methods and Encapsulation

## 6.1 Access modifiers

Il existe 4 manières de spécifier l'accès à un attribut d'une classe ou à une méthode :

1. `public` : peut être accédé depuis n'importe quelle classe
2. `private` : ne peut être accédé que depuis la même classe
3. `protected` : ne peut être accédé que par des classes qui sont dans le même paquet ou des sous–classes
4. *default* : ne peut être accédé que par des classes qui sont dans le même paquet

# 7 Exceptions

Les exceptions permettent d'indiquer que l'application a rencontré une erreur et qu'elle n'est pas capable de continuer le traitement en cours dû à cette erreur.

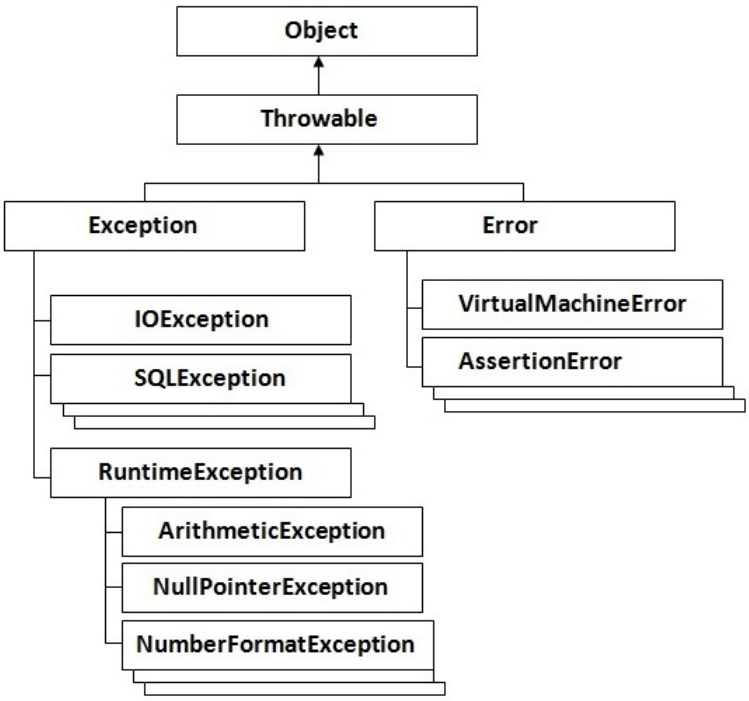


FIGURE 3 – Exceptions

## 7.1 java.lang.Error

Les erreurs sont très rarement, voire jamais jetées par le développeur d'une application. En général, les erreurs sont jetées par la JVM ; la plus connue étant la `java.lang.OutOfMemoryError` qui est jetée lorsque l'application n'a plus suffisamment de mémoire et crash.

## 7.2 Exceptions

Le développeur peut utiliser les exceptions déjà existantes dans l'API pour en jeter, soit en créer des spécifiques (bonne pratique).

### 7.2.1 Checked Exceptions

Les *checked exceptions* sont des exceptions qui lorsqu'elles sont jetées par une méthode **doivent** apparaître dans la signature de la méthode. De plus, les méthodes qui appelle cette dernière, **doivent** traiter cette exception ou la faire apparaître à leur tour dans leur signature.

Exemple :

```
public class MonException extends Exception {}

public void method1() throws MonException {
  throw new MonException();
}

public void method2() throws MonException {
  method1();
}

public void method3() {
  try {
  method1();
  } catch (MonException e) {
```

```
    // traiter l'exception si elle est jetée
  }
}
```

### 7.2.2 Unchecked Exceptions

Les *unchecked exceptions* sont des exceptions qui lorsqu'elles sont jetées par une méthode **ne sont pas obligées** apparaître dans la signature de la méthode. De plus, les méthodes qui appelle cette dernière, **ne sont pas obligées** de traiter cette exception ou la faire apparaître dans leur signature.

Exemple :

```
public class MonException extends RuntimeException {}

public void method1() {
  throw new MonException();
}

public void method2() {
  method1();
}

public void method3() throws MonException {
  method1();
}

public void method4() {
  try {
  method1();
  } catch (MonException e) {
    // traiter l'exception si elle est jetée
  }
}
```

## 7.3 Traiter une exception

Lorsqu'une exception est potentiellement jetée par une méthode que l'on invoque et qu'on souhaite la traiter, nous utilisons un bloc `try .. catch`.

Il est possible de spécifier plusieurs exceptions.
La première exception qui correspond sera utilisée.
Si on indique une exception parent, il n'est plus possible d'indiquer un traitement pour une sous–exception (si on souhaite le faire, il faut l'indiquer avant l'exception parent).

Le bloc `finally` permet d'indiquer un traitement à effectuer dans tous les cas (qu'une exception ait été jetée ou non).

Exemple :

```
public void method() {
  try {
  method1();
  } catch (IndexOutOfBoundsException e) {
    ...
  } catch (RuntimeException e) {
```

```
    ...
  } finally {
    ...
  }
}
```

Lorsque vous traiter une exception, vous avez la possibilité d'afficher le chemin qui a mené à cette exception en invoquant la méthode `e.printStackTrace()`. Ceci est très utile à des fins de debug.

## 7.4 Erreurs et Exceptions à connaître

### 7.4.1 Runtime Exceptions

API : `https://docs.oracle.com/javase/8/docs/api/java/lang/RuntimeException.html`

- `ArithmeticException` : lorsque l'on tente de faire une division par 0
- `ArrayIndexOutOfBoundsException` : lorsque l'on tente d'accèder à un index d'un tableau qui n'est pas dans la fourchette 0 *taille du tableau - 1*
- `ClassCastException` : lorsque l'on tente de caster un objet dans une classe dont l'objet n'est pas une instance
- `IllegalArgumentException` : indique à l'appelant qu'un paramètre n'est pas correct
- `NullPointerException` : lorsque l'on tente d'utiliser un objet qui est `null`
- `NumberFormatException` : lorsque l'on tente de parser un nombre à partir d'une chaîne de caractères qui n'est pas un nombre

### 7.4.2 Checked Exceptions

API : `https://docs.oracle.com/javase/8/docs/api/java/io/IOException.html`

- `FileNotFoundException` : lorsque l'on tente d'ouvrir un fichier qui n'existe pas
- `IOException` : lorsqu'une erreur de lecture/écriture apparaît (par exemple, si on tente d'écrire un fichier sur une clé USB et que la clé est retirée)

### 7.4.3 Erreurs

API : `https://docs.oracle.com/javase/8/docs/api/java/lang/Error.html`

- `ExceptionInInitializerError` : lorsqu'un bloc *static* d'initialisation jette une exception et ne la traite pas
- `StackOverflowError` : lorsqu'une méthode s'appelle elle–même et qu'il n'y a pas de condition d'arrêt ⇒ boucle infinie
- `NoClassDefFoundError` : lorsqu'un morceau de code fait appel à une classe qui n'est pas disponible à l'exécution (par exemple, fichier .class supprimé ou libraire manquante)